T3: Introduction to Data Analytics

Data Analytics with HPC
Master in High Performance Computing, Spring 2026

GRUPO ARQUITECTURA
DE COMPUTADORES
UNIVERSIDADE DA CORUÑA
GAC.UDC.ES

Contents

- 12/2 - Introduction to Python
  - Execution environments: iPython, Jupyter.
  - Basics.
  - Operators and types. Instructions. Functions. Modules. Classes. Exceptions.
  - Integrated tools.
  - NumPy.
  - Matplotlib.

Contents

- 19/2 - Data Analytics: Pandas
    - Intro to Pandas.
    - Data I/O.
    - Data wrangling.
    - Visualization.
    - Aggregation.
    - Time series.

Contents

- 26/2 - High level tools
    - Visualization: Seaborn.
    - Machine learning: Scikit-Learn
    - Out-of-core computing: Dask

Contents

- 5/3 - Projects **(online only)**
  - Time for developing your data analysis project.

Introduction to Python

Contents

- Execution environments: iPython, Jupyter.
- Basics.
- Operators and types. Instructions. Functions. Modules. Classes. Exceptions.
- Integrated tools.
- NumPy.
- Matplotlib.

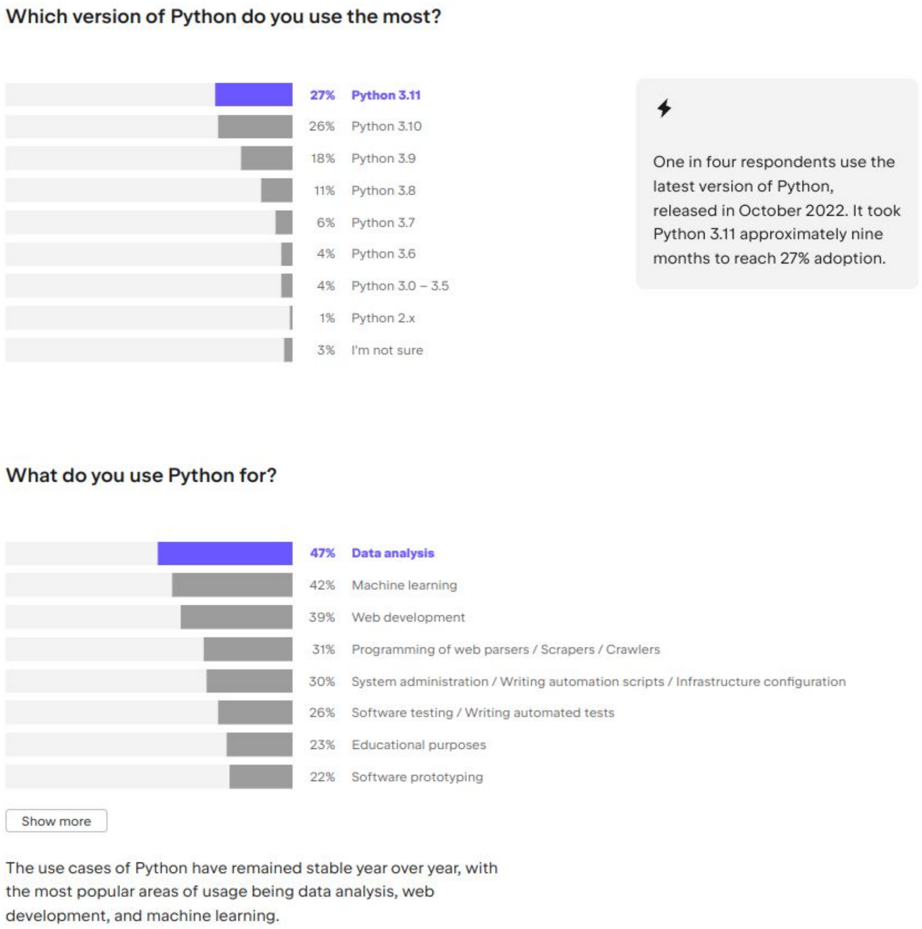
Python

- General-purpose language, interpreted, object-oriented.
- Portable: an interpreter exists for almost any OS.
- Open-source reference implementation (CPython), but alternative ones exist as well (e.g., iPython, Intel Python).
- Powerful and versatile:
  - Dynamic typing.
  - Complex “basic” types: lists, dictionaries, strings, ...
  - Built-in tools: powerful list operations, strings, arrays, ...
  - Libraries: math, statistics, parallelization, ...
  - Automatic memory management.

Python

- Popular with the scientific and engineering communities.
- Free and general purpose alternative to Matlab/R.
- “Slice” notation (Matlab-like).
- Multiple extensions: NumPy, Matplotlib, SciPy, pytables, …
- Simple integration: C/C++/Fortran, web services, …
- Web development: Django, Flask, Pyramid, …
- Scientific computing environments: iPython, conda-forge, …

Python: “state of the art”



Which version of Python do you use the most?

27% Python 3.11
26% Python 3.10
18% Python 3.9
11% Python 3.8
6% Python 3.7
4% Python 3.6
4% Python 3.0 – 3.5
1% Python 2.x
3% I'm not sure

One in four respondents use the latest version of Python, released in October 2022. It took Python 3.11 approximately nine months to reach 27% adoption.

What do you use Python for?

47% Data analysis
42% Machine learning
39% Web development
31% Programming of web parsers / Scrapers / Crawlers
30% System administration / Writing automation scripts / Infrastructure configuration
26% Software testing / Writing automated tests
23% Educational purposes
22% Software prototyping

The use cases of Python have remained stable year over year, with the most popular areas of usage being data analysis, web development, and machine learning.

Source: https://www.jetbrains.com/lp/devecosystem-2023/python/

iPython

- Interactive Python interpreter.
- Does not provide analytics or computing tools, but greatly eases up the use of Python and the development process.
- Includes GUIs for easy data visualization.
- Allows to work remotely through a web browser, as well as storing work sessions in an HTML format to allow sharing code, data, and examples.

Jupyter Notebooks

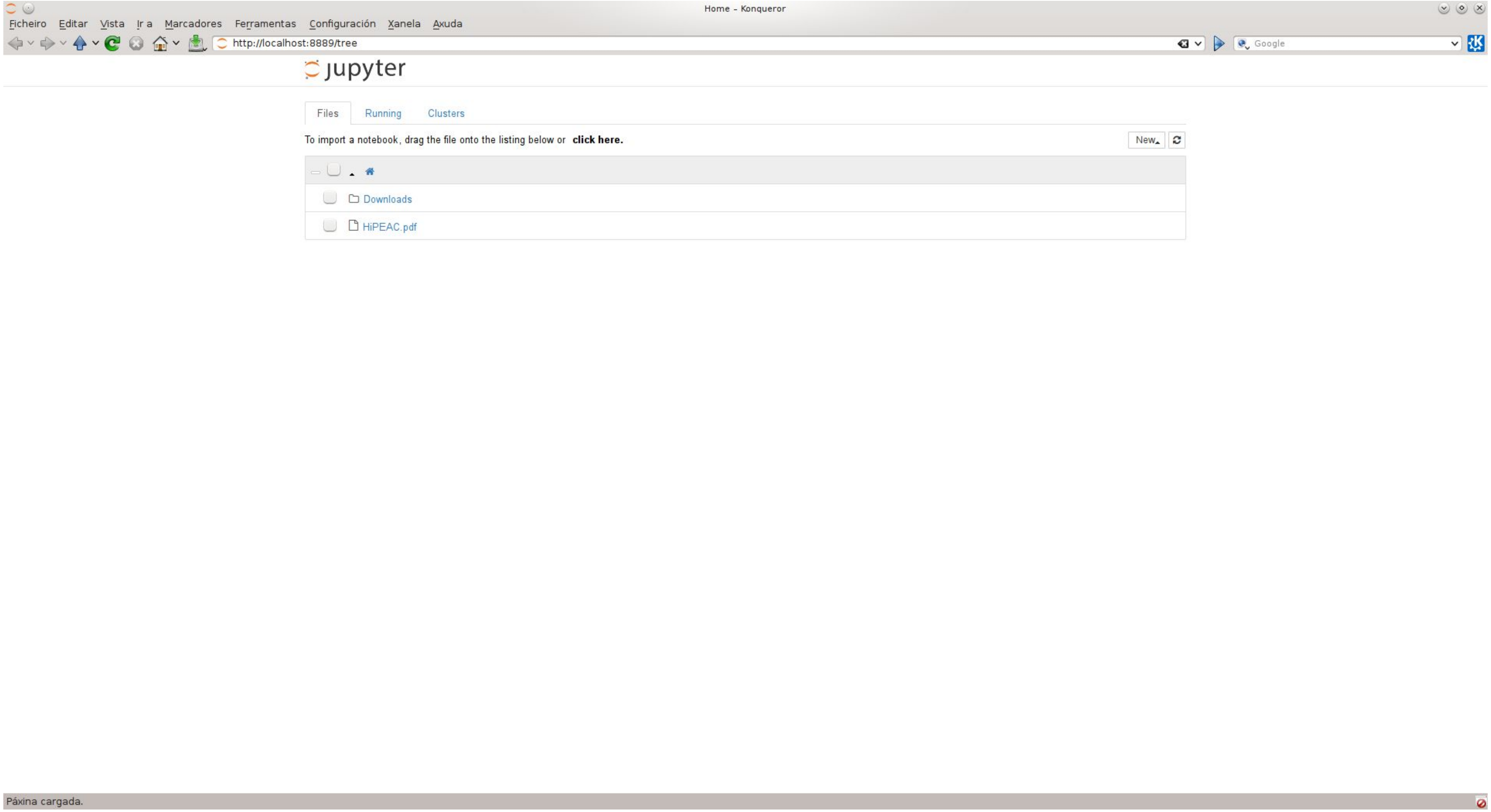
- Environment focused on exporting HTML or being used interactively.

- Uses a JSON-based format to allow sharing code and outputs.

- Built as a web server to which clients are connected using a browser.

- Allows to remotely use a Python environment via web.

Jupyter Notebooks



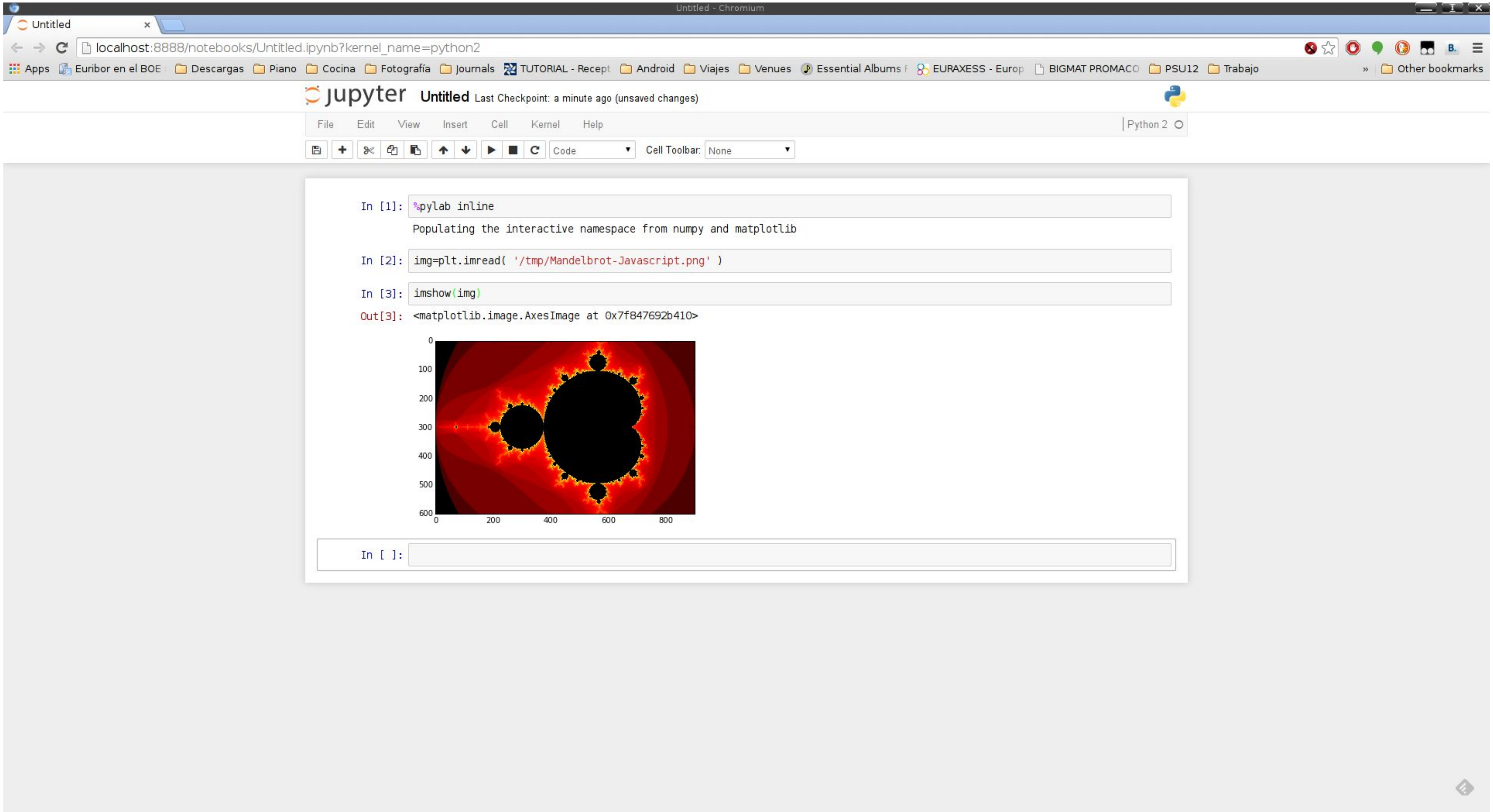
Home - Konqueror
Ficheiro Editar Vista Ir a Marcadores Ferramentas Configuración Xanela Axuda
http://localhost:8889/tree
Google
jupyter
Files
Running
Clusters
New
Páxina cargada.

To import a notebook, drag the file onto the listing below or **click here.**

- Downloads
- HiPEAC.pdf

01-Example.ipynb

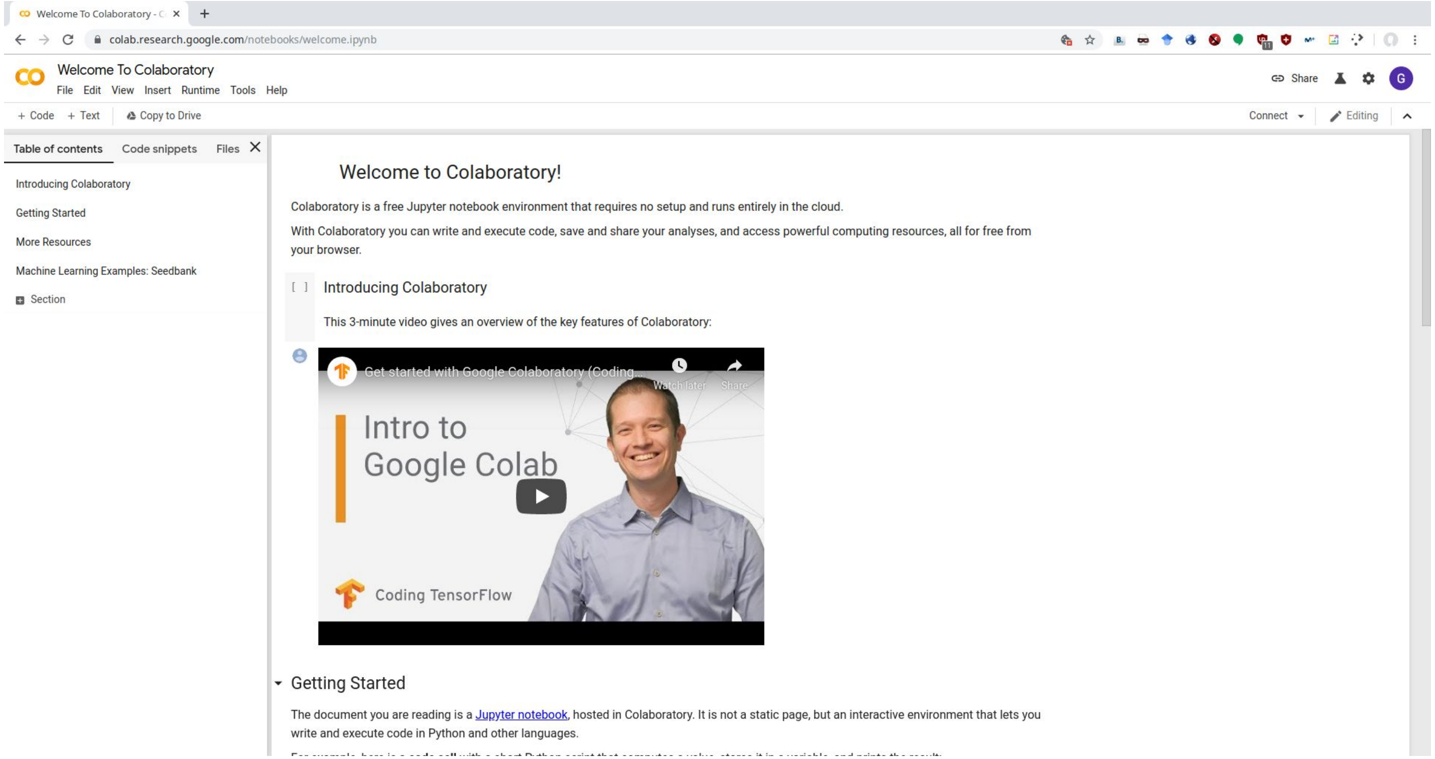
Jupyter Notebooks



```
In [1]: %pylab inline
        Populating the interactive namespace from numpy and matplotlib

In [2]: img=plt.imread( '/tmp/Mandelbrot-Javascript.png' )

In [3]: imshow(img)
Out[3]: <matplotlib.image.AxesImage at 0x7f847692b410>

In [ ]:
```

Google Colaboratory

- Free notebook environment on the cloud.
- https://colab.research.google.com


Welcome To Colaboratory - C
colab.research.google.com/notebooks/welcome.ipynb
Welcome To Colaboratory
File Edit View Insert Runtime Tools Help
Share
G
+ Code + Text
Copy to Drive
Connect
Editing
Table of contents Code snippets Files
Introducing Colaboratory
Getting Started
More Resources
Machine Learning Examples: Seedbank
Section
Get started with Google Colaboratory (Coding...
Watch later
Share
Intro to Google Colab
Coding TensorFlow

Welcome to Colaboratory!

Colaboratory is a free Jupyter notebook environment that requires no setup and runs entirely in the cloud.

With Colaboratory you can write and execute code, save and share your analyses, and access powerful computing resources, all for free from your browser.

Introducing Colaboratory

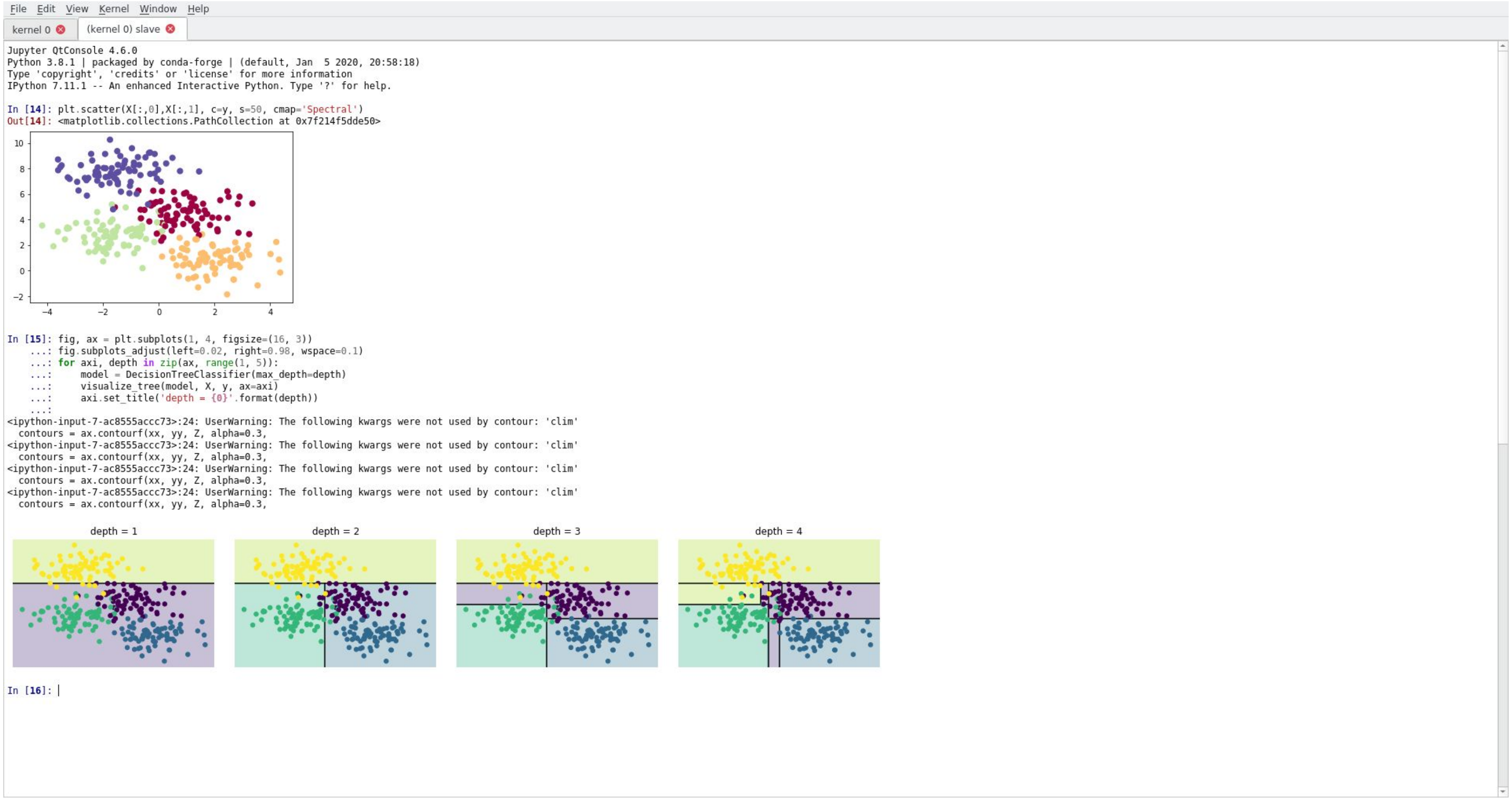
This 3-minute video gives an overview of the key features of Colaboratory:

Getting Started

The document you are reading is a Jupyter notebook, hosted in Colaboratory. It is not a static page, but an interactive environment that lets you write and execute code in Python and other languages.

For example, here is a **code cell** with a short Python script that computes a value, stores it in a variable, and prints the result:

Jupyter QtConsole

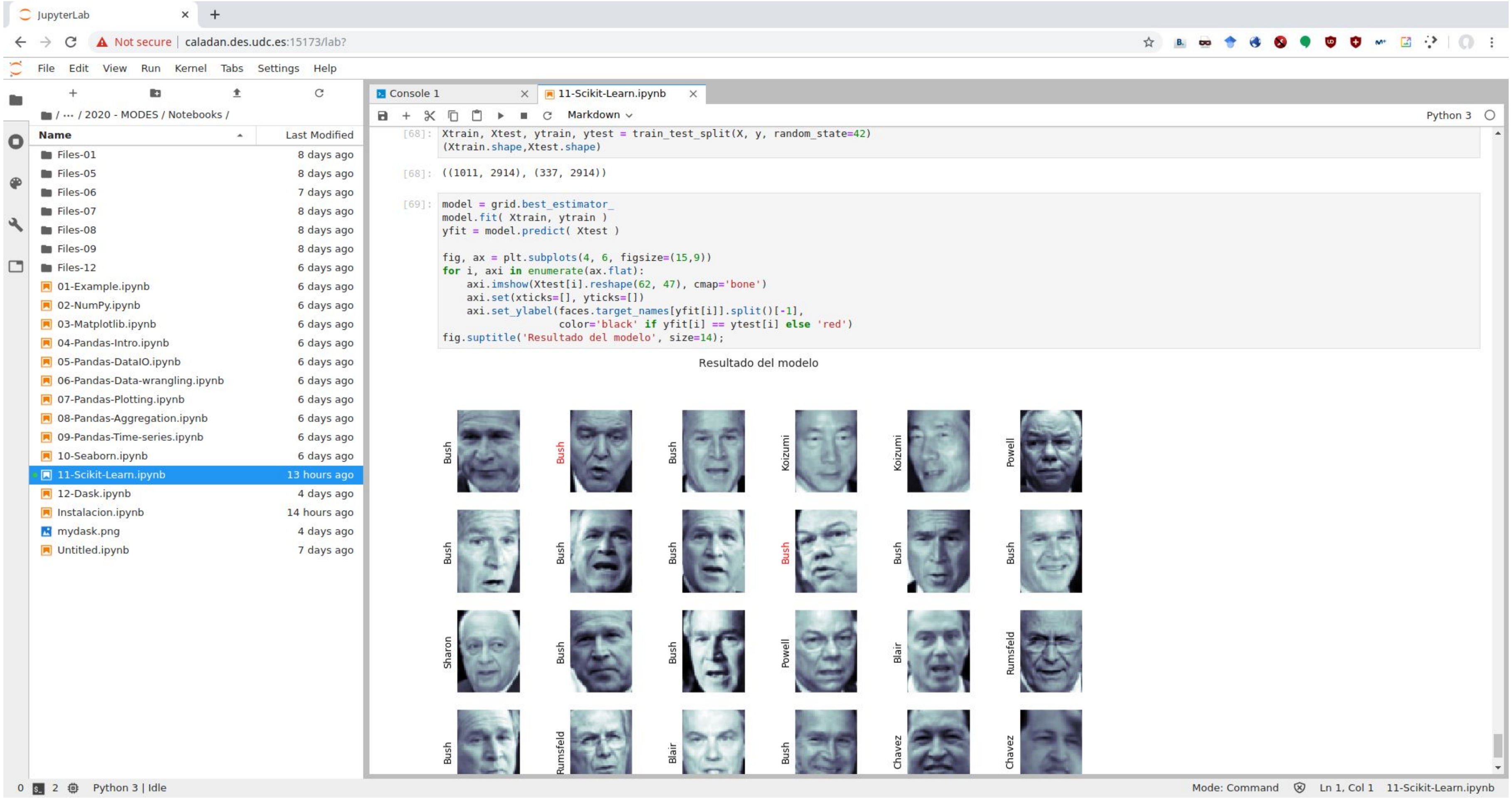


```
Jupyter QtConsole 4.6.0
Python 3.8.1 | packaged by conda-forge | (default, Jan  5 2020, 20:58:18)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.11.1 -- An enhanced Interactive Python. Type '?' for help.

In [14]: plt.scatter(X[:,0],X[:,1], c=y, s=50, cmap='Spectral')
Out[14]: <matplotlib.collections.PathCollection at 0x7f214f5dde50>
```

```
In [15]: fig, ax = plt.subplots(1, 4, figsize=(16, 3))
    ...: fig.subplots_adjust(left=0.02, right=0.98, wspace=0.1)
    ...: for axi, depth in zip(ax, range(1, 5)):
    ...:     model = DecisionTreeClassifier(max_depth=depth)
    ...:     visualize_tree(model, X, y, ax=axi)
    ...:     axi.set_title('depth = {0}'.format(depth))
    ...:
<ipython-input-7-ac8555accc73>:24: UserWarning: The following kwargs were not used by contour: 'clim'
  contours = ax.contourf(xx, yy, Z, alpha=0.3,
<ipython-input-7-ac8555accc73>:24: UserWarning: The following kwargs were not used by contour: 'clim'
  contours = ax.contourf(xx, yy, Z, alpha=0.3,
<ipython-input-7-ac8555accc73>:24: UserWarning: The following kwargs were not used by contour: 'clim'
  contours = ax.contourf(xx, yy, Z, alpha=0.3,
<ipython-input-7-ac8555accc73>:24: UserWarning: The following kwargs were not used by contour: 'clim'
  contours = ax.contourf(xx, yy, Z, alpha=0.3,
```

```
In [16]: |
```

JupyterLab: Next Generation Notebook Interface


JupyterLab
Not secure | caladan.des.udc.es:15173/lab?
File Edit View Run Kernel Tabs Settings Help
/ ··· / 2020 - MODES / Notebooks /
Console 1
11-Scikit-Learn.ipynb
Markdown
Python 3

Name	Last Modified
Files-01	8 days ago
Files-05	8 days ago
Files-06	7 days ago
Files-07	8 days ago
Files-08	8 days ago
Files-09	8 days ago
Files-12	6 days ago
01-Example.ipynb	6 days ago
02-NumPy.ipynb	6 days ago
03-Matplotlib.ipynb	6 days ago
04-Pandas-Intro.ipynb	6 days ago
05-Pandas-DataIO.ipynb	6 days ago
06-Pandas-Data-wrangling.ipynb	6 days ago
07-Pandas-Plotting.ipynb	6 days ago
08-Pandas-Aggregation.ipynb	6 days ago
09-Pandas-Time-series.ipynb	6 days ago
10-Seaborn.ipynb	6 days ago
11-Scikit-Learn.ipynb	13 hours ago
12-Dask.ipynb	4 days ago
Instalacion.ipynb	14 hours ago
mydask.png	4 days ago
Untitled.ipynb	7 days ago

```
[68]: Xtrain, Xtest, ytrain, ytest = train_test_split(X, y, random_state=42)
      (Xtrain.shape,Xtest.shape)

[68]: ((1011, 2914), (337, 2914))

[69]: model = grid.best_estimator_
      model.fit( Xtrain, ytrain )
      yfit = model.predict( Xtest )

      fig, ax = plt.subplots(4, 6, figsize=(15,9))
      for i, axi in enumerate(ax.flat):
          axi.imshow(Xtest[i].reshape(62, 47), cmap='bone')
          axi.set(xticks=[], yticks=[])
          axi.set_ylabel(faces.target_names[yfit[i]].split()[-1],
                         color='black' if yfit[i] == ytest[i] else 'red')
      fig.suptitle('Resultado del modelo', size=14);
```

Resultado del modelo

Python: Conceptual hierarchy

- Programs are made of **modules**.
- Modules contain **functions**.
- Functions contain **instructions**.
- Instructions contain **expressions**.
- Expressions create and manipulate **objects**.
- Objects belong to a **class.**

Python: Conceptual hierarchy

```
>>>>>>>>> file.py

def f():
    i = 0

    while i < 100:
        print( i )
        i += 1
```

Python: Objects

- **Everything** is an object:
    - integer numbers -> *int*
    - floating point numbers -> *float*
    - character strings -> *str*
    - lists -> *list*
    - dictionaries -> *dict*
- The type of a variable can be queried using *type()*.
- Objects are containers which aggregate variables (*attributes*) and functions (*methods*).
- Objects belong to a *class*, which is an abstract description (or scheme) of the common traits of a family of objects.

Python - Executing programs

1. Using the interactive interpreter:

```
>>> txt = “Luke, I am your father”

>>> print( txt )

Luke, I am your father
```

Python - Executing programs

2. Interpreting Python code:

```
------------------------------- > script.py

txt = “Luke, I am your father”

print( txt )

-------------------------------
```

```
$ python script.py

Luke, I am your father
```

Python - Executing programs

3. Shell script:

```
------------------------------- > script.py
#! /usr/bin/python
txt = “Luke, I am your father”
print( txt )
-------------------------------
```

```
$ ./script.py
Luke, I am your father
```

Python - Executing programs

4. Embedding Python into C:

```
------------------------------- > test.c

#include <Python.h>

…

Py_Initialize();

PyRun_SimpleString( “txt = ‘Luke, I am your father’” );

PyRun_SimpleString( “print( txt )” );

…

-------------------------------

$ gcc -I /usr/include/python3.8 test.c -lpython3.8

$ ./a.out

Luke, I am your father
```

Python - Semantics & Syntax

Indentation, not brackets

```
if x < threshold:

    print( “x less than threshold” )

else:

    print( “x greater than threshold” )
```

Python - Semantics & Syntax
Everything is an object

- Including numbers, strings, functions, classes, modules, ...
- Each object has a type and its own data.
- This provides flexibility, allowing to deal with any element in a generic way.

Python - Semantics & Syntax
Comments

```
# This is a dramatic moment.

txt = “Luke, I am your father”

print( txt )
```

Python - Semantics & Syntax
Function calls

- C syntax:

```
result = f( x, y, z )
```

- Class methods:

```
result = obj.f( x, y, z )
```

- Functions may be passed positional or keyword arguments:

```
result = f( x, y, z, tol=0.01, method=”fast” )
```

Python - Semantics & Syntax
Variables and passing by reference

- Assignment to a variable generates a new reference to an object.
- The original object remains the same!

```
>>> a = [1,2,3]

>>> b = a

>>> a.append(4)

>>> b

[1,2,3,4]
```

Python - Semantics & Syntax
Variables and passing by reference

Consequently:

- Passing large variables is efficient.
- ¡Functions may permanently modify their parameters!
- Except for **immutable** types. E.g.,

```
>>> a = 10
>>> b = a
>>> a = 20
>>> print b
10
```

Python - Semantics & Syntax
Dynamic references, strong typing

References (variables) have no static type:

```
>>> a = 10

>>> type(a)

<type ‘int’>

>>> a = “10”

>>> type(a)

<type ‘str’>
```

Python - Semantics & Syntax
Dynamic references, strong typing

Python has strong typing, nevertheless:

```
>>> 5 + “5”
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> “5” + 5
  TypeError: cannot concatenate 'str' and 'int' objects
  >>> 5 + int(“5”)
  10
  >>> “5” + str(5)
  ‘55’
```

Each Python object has a specific type (class). Implicit casting is only performed for `int` to floating point conversions.

Python - Semantics & Syntax
Dynamic references, strong typing

The `isinstance` operator is used to check whether an object belongs to a particular class:

```
>>> a = 4.5

>>> isinstance(a,int)

False

>>> isinstance(a,float)

True
```

Python - Semantics & Syntax
Methods and Attributes

Python objects have methods and attributes:

```
>>> a = 4.5

>>> dir(a)

['__abs__', '__add__', '__class__', '__coerce__', '__delattr__', '__div__', '__divmod__',
'__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__ge__',
'__getattribute__', '__getformat__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__int__', '__le__', '__long__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__',
'__new__', '__nonzero__', '__pos__', '__pow__', '__radd__', '__rdiv__', '__rdivmod__',
'__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__',
'__rpow__', '__rsub__', '__rtruediv__', '__setattr__', '__setformat__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', 'as_integer_ratio',
'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

Python - Semantics & Syntax
Methods and Attributes

```
>>> a.hex()

'0x1.2000000000000p+2'

>>> hasattr( a, “hex” )

True

>>> getattr( a, “hex” )

<built-in method hex of float object at 0x1ed86a0>

>>> getattr( a, “hex” )()

'0x1.2000000000000p+2'
```

Python - Semantics & Syntax
Importing modules

- A Python module is a `.py` file including definitions and/or code:

```
------------------------------- > modulo1.py

PI = 3.14159


def f(x):

    return x + 2


def g(a,b):

    return a + b

-------------------------------
```

Python - Semantics & Syntax
Binary operators

`a + b`	Addition	`a - b`	Subtraction
`a * b`	Multiplication	`a / b`	Division (Python 3)
`a // b`	Integer division (Python 3)	`a ** b`	Exponentiation
`a & b`	Bitwise AND	`a | b`	Bitwise OR
`a ^ b`	Bitwise XOR	`a == b`	Equality
`a != b`	Inequality	`a < b, a <= b`	Less/Less or equal than
`a > b, a >= b`	Greater/Greater or equal than	`a is b`	Identity: True if `a` and `b` are the same object

Python - Scalar types

`None`	“null” in Python. Singleton class.
`str`	String type. ASCII in Python 2.x, Unicode in Python 3.
`unicode`	Unicode string.
`float`	Floating point number, double precision (64 bits).
`bool`	Logical, `True` or `False`.
`int`	Signed integer. 32 or 64 bits depending on platform. Arbitrary precision in Python 3.
`long`	Signed integer, arbitrary precision. Does not exist in Python 3.

Python - Scalar types
Numerical types

- Python 2: `int` (32- or 64-bits), `float` (64 bits) and `long` (arbitrary precision).
- Python 3: `int` (arbitrary precision), `float` (64 bits).
- Conversion between `int` and `long` is transparent in Python 2.
- Scientific notation accepted:

```
>>> fval = 6.78e-5
```

Python - Scalar types
Numerical types

- Division of two integers in Python 2 yields an integer (truncated, not rounded). For floating point division of integers in Python 2:

```
>>> fval = a_int / float(b_int)
```

- Division of integers in Python 3 yields a floating point number. For integer division of integers in Python 3:

```
>>> ival = a_int // b_int
```

- Complex numbers are written using `j` for the imaginary part:

```
>>> cval = 1 + 2j
```

Python - Scalar types

Strings

- String literals are written between single or double quotes:

```
>>> a = ‘one way to write a string’

>>> b = “a different way”
```

- To write strings with line breaks, use three single quotes:

```
>>> c = ‘‘‘

This is a larger string

Which spans more than one line

’’’
```

Python - Scalar types

Strings

- Strings are immutable objects: once they are created, they cannot be modified:

```
>>> a = ‘this is a string’

>>> a[10] = ‘f’

TypeError: ‘str’ object does not support item assignment

>>> b = a.replace( ‘string’, ‘larger string’)

>>> b

‘this is a larger string’
```

Python - Scalar types

Strings

- Strings can be created from different types using `str()`:

```
>>> a = 5.6

>>> s = str(a)

>>> s

‘5.6’
```

Python - Scalar types

Strings

- A string is just a **collection** of characters (so not really a scalar type):

```
>>> s = ‘python’

>>> list(s)

[ ‘p’, ‘y’, ‘t’, ‘h’, ‘o’, ‘n’ ]

>>> s[:3]

‘pyt’
```

Python - Scalar types

Strings

- The backslash (`\`) is used as the scape character, to encode special characters such as line breaks (`\n`) or Unicode characters.
- A string which should be intepreted literally (raw) can be marked using `r‘string’`:

```
>>> s = r‘no\special\characters\in\this\string’

>>> s

‘no\special\characters\in\this\string’
```

Python - Scalar types

Strings

- Operator + applies string concatenation:

```
>>> a = ‘first half of string ’

>>> b = ‘and second half of string’

>>> a + b

‘First half of string and second half of string’
```

Python - Scalar types

Strings

- To specify that a string is encoded using Unicode, `u’string’` is used.
- To format a string there’s a C-like template syntax:

```
>>> template = u“%.2f %s are worth %d€”

>>> template % (1.18018, u‘american dollars’, 1)

u'1.18 american dollars are worth 1\u20ac'

>>> print template % (1.18018, u‘american dollars’, 1)

‘1.18 american dollars are worth 1€’
```

Python - Scalar types

Booleans

- The two boolean values are written `True` and `False`.
- They can be combined using the keywords `and` and `or`.

```
>>> True and True

True

>>> False and True

True
```

Python - Scalar types

Booleans

- All basic predefined types in Python, as well as any class implementing the `__nonzero__()` method, can be interpreted as `True` or `False` in a conditional clause.

```
>>> a = [1,2,3]

>>> if a: print “True”

‘True’

>>> b = []

>>> if not b: print “False”

‘False’
```

Python - Scalar types
None

- None is the null value in Python.
- If a function does not explicitly return a value, it returns None implicitly.
- Can be used as a default value for optional parameters to functions:

```
def add_and_maybe_multiply( a, b, c = None ):

    result = a + b

    if c is not None: result *= c

    return result
```

- None is not a reserved word, but an instance (singleton) of NoneType.

Python - Scalar types
Dates and times

- The `datetime` provides the types `datetime`, `date` and `time`.

```
>>> from datetime import datetime, date, time

dt = datetime( 2015, 04, 24, 12, 25, 32 )
```

```
>>> dt.day

24
```

```
>>> dt.time()

datetime.time( 12, 25,
32 )
```

Python - Scalar types

Dates and times

- The `strftime()` method formats a `datetime` object as a string:

```
>>> dt.strftime( ‘%d/%m/%Y %H:%M’ )

‘24/04/2015 12:25’
```

- A `datetime` object can be created from a string using `strptime`:

```
>>> datetime.strptime( ‘20150424’, ‘%Y%m%d’ )

datetime.datetime( 2015, 4, 24, 0, 0 )
```

Python - Scalar types
Dates and times

- The subtraction of two `datetime` objects returns a `datetime.timedelta`:

```
>>> dt2 = datetime( 2015, 4, 25, 12, 25, 32 )

>>> dt2 - dt

datetime.timedelta(1)
```

- Operating a `datetime` object with a `timedelta` returns a new `datetime`:

```
>>> from datetime import timedelta

>>> dt + timedelta(days=1)

>>> datetime.datetime(2015, 4, 25, 12, 25, 32)
```

Python - Flow control

if, elif, else

These blocks behave like in most other languages:

```
if x < 0:
    print ‘Negative’
elif x == 0:
    print ‘Zero’
elif 0 < x < 5:
    print ‘Positive less than 5’
else:
    print ‘Positive and greater or equal than 5’
```

Python - Flow control

`for` loops

- A `for` loop is used to iterate over a collection (like a list or tuple):

```
for val in collection:

    # Do something with val
```

- `continue` skips to the next iteration
- `break` exits the current loop

Python - Flow control

`range()` and `xrange()`

- `range()` returns a list of equally spaced integers:

```
>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Takes as parameters the start, end, and step of the sequence:

```
>>> range(5,20,3)

[5,8,11,14,17]
```

- Generates values in the interval `[start,end).`

Python - Flow control

`range()` and `xrange()`

- `range()` is useful to code `for` loops with C semantics:

```
>>> for x in range(10):
...     print 2*x
0
2
4
8
16
18
```

Python - Flow control

range() and xrange()

- xrange() accepts the same parameters as range(), but returns a generator instead of building the full list in memory.
- It features lazy evaluation.

```
>>> xrange(10)

xrange(10)

>>> for x in xrange(10):

...     print 2*x

0

2

…
```

- It is preferable to use xrange() when working with large ranges.

Python - Flow control

`while` loops

- Iterate **while** a condition is met:

```
x = 256

total = 0

while x > 0:

    if total > 500: break

    total += x

    x = x // 2
```

Python - Flow control

pass

- `pass` is a no-op instruction in Python.
- It is sometimes required because white spaces in Python delimit execution blocks:

```
if x < 0:

    print “Negativo”

elif x == 0: pass

else:

    print “Positivo”
```

Python - Flow control
Exception handling

- Functions and operations can raise exceptions:

```
>>> 5 / 0

ZeroDivisionError: integer division or modulo by zero
```

- Exceptions can be handled to solve runtime errors.
- This allows to dynamically manage some selected error types.

Python - Flow control

Exception handling

```
def float_division( a, b ):
    x = NaN
    try:
        x = a / float(b)
    except:
        print “Exception during division”
    return x
```

Python - Flow control
Exception handling

- We can explicitly list the exception types managed by a `except` block:

```
def float_division( a, b ):
    x = NaN
    try:
        x = a / float(b)
    except ZeroDivisionError:
        print “Division by zero”
    return x
```

Python - Flow control
Exception handling

- Multiple exception types can be managed by the same block:

```
def float_division( a, b ):
    x = NaN
    try:
        x = a / float(b)
    except ZeroDivisionError, TypeError:
        print “Exception during division”
    return x
```

Python - Flow control
Exception handling

- A finally block is executed regardless of whether the try was successful:

```
f = open( path, ‘w’ )

try:

    write_to_file( f )

finally:

    f.close()
```

Python - Flow control
Exception handling

- An `else` block will only be executed if the `try` was successful:

```
f = open( path, ‘w’ )

try:

    write_to_file( f )

except: print ‘Failure’

else: print ‘Success’

finally: f.close()
```

Python - Collections

Tuples

- Tuple: unidimensional sequence of fixed size containing Python objects.

```
>>> tup = (2, 3, 4)

>>> nested_tup = ( (1, 2), (2, 3, 4) )

>>> tup_from_list = tuple( [2,3,4] )

>>> tup_from_iter = tuple( xrange(5) )

>>> tup_from_string = tuple( ‘string’ )
```

Python - Collections

Tuples

- The elements in a tuple are accessed using the `[]` operator:

```
>>> tup_from_iter[3]

3

>>> tup_from_iter[1:3]

(1,2)

>>> tup_from_string[0]

‘s’
```

Python - Collections

Tuples

- Tuples are immutable:

```
>>> tup_from_iter[3] = 5

TypeError: ‘tuple’ object does not support item assignment
```

- However, objects inside a tuple can be mutable:

```
>>> tup = ( (1,2), [3,4,5] )

>>> tup[1].append(6)

>>> tup

((1, 2), [3, 4, 5, 6])
```

Python - Collections

Tuples

- Tuples are concatenated using the + operator:

```
>>> tup_from_iter + tup_from_string

(0, 1, 2, 3, 4, ‘s’, ‘t’, ‘r’, ‘i’, ‘n’, ‘g’)
```

- The * operator is consistent with the addition/concatenation semantics:

```
>>> tup_from_iter*3

(0, 1, 2, 3, 4, 0, 1, 2, 3, 4, 0, 1, 2, 3, 4)
```

Python - Collections

Tuples

- Note that the objects contained in tuples are not copied, but referenced:

```
>>> tup = ( (1, 2), (3, 4, 5) )

>>> tup2 = tup * 3

>>> tup[1].append(6)

>>> tup2

((1, 2), [3, 4, 5, 6], (1, 2), [3, 4, 5, 6], (1, 2), [3, 4,
5, 6])
```

Python - Collections

Tuples

- If the right hand side of an assignment is a tuple, Python tries to unpack it:

```
>>> tup = (1,2,3)

>>> a, b, c = tup

>>> b

2
```

Python - Collections

Tuples

- Nested tuples can be explicitly unpacked:

```
>>> tup = ( 1, 2, (3, 4) )

>>> a, b, (c, d) = tup

>>> c

3

>>> a, b = b, a

>>> a

2
```

- A common use of tuples is to code functions that return multiple values.

Python - Collections
Lists

- Unlike tuples, lists have variable length and are mutable:

```
>>> list_1 = [2, 3, 7, None]

>>> tup = ( ‘a’, ‘b’, ‘c’ )

>>> list_2 = list( tup )

>>> list_2[1] = ‘d’

>>> list_2

[ ‘a’, ‘d’, ‘c’ ]
```

Python - Collections

Lists

- Elements are added at the end of a list using `append()`:

```
>>> list_2.append( 5 )

>>> list_2

[ ‘a’, ‘d’, ‘c’, 5 ]
```

- Elements can also be added at a specific place using `insert()`:

```
>>> list_2.insert( 3, None )

>>> list_2

[ ‘a’, ‘d’, ‘c’, None, 5 ]
```

Python - Collections
Lists

- The reverse operation to `insert()` is `pop()`:

```
>>> list_2.pop()

5

>>> list_2

[ ‘a’, ‘d’, ‘c’, None ]

>>> list_2.pop(2)

‘c’

>>> list_2

[ ‘a’, ‘d’, None ]
```

Python - Collections

Lists

- Elements can be deleted from the list using `remove()`:

```
>>> list_2.append(‘a’)
>>> list_2
[ ‘a’, ‘d’, None, ‘a’ ]
>>> list_2.remove( ‘a’ )
>>> list_2
[ ‘d’, None, ‘a’ ]
```

Python - Collections

Lists

- The `in` operator checks whether a value is contained in a list:

```
>>> ‘a’ in list_2

True

>>> ‘c’ in list_2

False
```

Python - Collections

Lists

- The operator `+` is used to concatenate lists:

```
>>> list_1 + list_2

[ 2, 3, 7, None, ‘d’, None, ‘a’ ]
```

- `extend()` adds full lists to a given one:

```
>>> list_1.extend( list_2 )

>>> list_1

[ 2, 3, 7, None, ‘d’, None, ‘a’ ]
```

Python - Collections
Lists

- A list can be sorted in place using `sort()`:

```
>>> a = [ 7, 2, 5, 1, 3 ]

>>> a.sort()

>>> a

[1, 2, 3, 5, 7]

>>> b = [‘galicia’, ‘asturias’, ‘cantabria’, ‘euskadi’, ‘navarra’]

>>> b.sort()

>>> b

[ ‘asturias’, ‘cantabria’, ‘euskadi’, ‘galicia’, ‘navarra’ ]
```

Python - Collections
Lists

- `sort()` accepts a function as sorting key:

```
>>> b = [ ‘caladan’, ‘arrakis’, ‘corrin’, ‘ix’, ‘giedi
prime’ ]

>>> b.sort( key = len )

>>> b

[ ‘ix’, ‘corrin’, ‘caladan’, ‘arrakis’, ‘giedi prime’
]
```

Python - Collections

Lists

- The `bisect` module manipulates sorted lists using binary search.
- It does not check that a list is actually sorted, using it on unsorted lists yields incorrect results.

```
>>> import bisect

>>> c = [1, 2, 2, 2, 3, 4, 7]

>>> bisect.bisect( c, 2 ) # Returns insertion index

4

>>> bisect.insort( c, 6 ) # Sorted insertion

>>> c

[1, 2, 2, 2, 3, 4, 6, 7]
```

Python - Collections

Slicing

- Sections of indexed collections (such as tuples and lists) can be accessed using slice notation:

```
>>> a = [7, 2, 3, 7, 5, 6, 0, 1]

>>> a[1:5]

[2, 3, 7, 5]



>>> a[3:4] = [6, 3]

>>> a

[7, 2, 3, 6, 3, 5, 6, 0, 1]
```

Python - Collections

Slicing

- The slice `[start:stop]` includes the element in the `start` position, but not the element in the `stop` position.
- Any of them may be omitted, in which case the start or ending element of the collection is used by default:

```
>>> a[:5]

[7, 2, 3, 6, 3]

>>> a[3:]

[6, 3, 5, 6, 0, 1]
```

Python - Collections

Slicing

- Negative indices refer to the end of the array:

```
>>> a[-4:]

[5, 6, 0, 1]

>>> a[-6:-2]

[6, 3, 5, 6]
```

Python - Collections

Slicing

- The step can be modified using `[start:stop:step]`:

```
>>> a[::2]

[7, 3, 3, 6, 1]
```

- A particular use of the step is the reverse a sequence:

```
>>> a[::-1]

[1, 0, 6, 5, 3, 6, 3, 2, 7]
```

Python - Collections Manipulation

- Oftentimes we want to iterate the elements of a collection and their index at the same time:

```
i = 0

for x in collection:

    # do something with x, i

    i += 1
```

- This is equivalent to

```
for i, x in enumerate(collection):

    # do something with x, i
```

Python - Collections Manipulation

- `sorted()` returns a sorted list containing the elements of a collection:

```
>>> sorted( [7, 1, 2, 6, 0, 3, 2] )

[0, 1, 2, 2, 3, 6, 7]

>>> sorted( ‘test string’ )

[' ', 'e', 'g', 'i', 'n', 'r', 's', 's', 't', 't',
't']

>>> sorted( set( ‘test string’ ) )

[' ', 'e', 'g', 'i', 'n', 'r', 's', 't']
```

Python - Collections Manipulation

- `zip()` groups the elements in several sequences into a list of tuples:

```
>>> seq1 = [‘caladan’, ‘kaitain’, ‘giedi prime’, ‘arrakis’]

>>> seq2 = [‘atreides’, ‘corrino’, ‘harkonnen’]

>>> zip( seq1, seq2 )

[(‘caladan’, ‘atreides’), (‘kaitain’, ‘corrino’),

(‘giedi prime’, ‘harkonnen’)]
```

- The length of the resulting list is given by the length of the shortest input sequence.

Python - Collections Manipulation

- `zip()` is commonly used to simultaneously iterate several sequences:

```
for i, (a,b) in enumerate( zip( seq1, seq2 ) ):
    # do something with i, a, b
```

Python - Collections Manipulation

- `zip()` can be used to “unzip”:

```
>>> l = [ (‘caladan’, ‘atreides’), (‘kaitain’, ‘corrino’),
(‘giedi prime’, ‘harkonnen’) ]

>>> unzip1, unzip2 = zip( *l )

>>> unzip1

(‘caladan’, ‘kaitain’, ‘giedi prime’)

>>> unzip2

(‘atreides’, ‘corrino’, ‘harkonnen’)
```

Python - Collections Manipulation

- `reversed()` builds an iterator over the elements of a sequence in reverse order:

```
>>> reversed( range(10) )

<listreverseiterator object at 0x7ff664159510>

>>> list( reversed( range(10) ) )

[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Python - Collections Dictionaries

- A dictionary (`‘dict’` type) is an associative array (hashtable).
- It is similar to an array, but indexes an object using a key instead of an integer.

```
>>> empty_dict = {}

>>> d1 = { ‘a’: ‘a value’, ‘b’: [1,2,3,4] }

>>> d1

{‘a’: ‘a value’, ‘b’: [1,2,3,4]}
```

Python - Collections

Dictionaries

- The elements of a dictionary can be accessed and inserted using the same syntax as for lists and tuples:

```
>>> d1[7] = ‘an integer’

>>> d1

{‘a’: ‘a value’, ‘b’: [1,2,3,4], 7: ‘an integer’}

>>> d1[‘b’]

[1,2,3,4]
```

Python - Collections

Dictionaries

- It can be checked whether a key is contained in a dictionary using `in`:

```
>>> ‘b’ in d1

True

>>> 42 in d1

False
```

Python - Collections
Dictionaries

- To remove values from a dictionary either `del` or `pop()` can be used:

```
>>> del d1['a']

>>> d1

{'b': [1,2,3,4], 7: 'an integer'}

>>> d1.pop('b')

[1,2,3,4]

>>> d1

{7: 'an integer'}
```

Python - Collections
Dictionaries

- `keys()` and `values()` return the keys and values stored in the dictionary:

```
>>> d1[5] = ‘another integer’

>>> d1.keys()

[5,7]

>>> d1.values()

[‘another integer’, ‘an integer’]
```

- The keys and values are not returned in any particular order, but the orders of both lists are consistent (i.e. `d1[d1.keys()[x]] = d1.values()[x]`).

Python - Collections
Dictionaries

- Two dictionaries can be fused using `update()`:

```
>>> d1.update( { ‘b’: ‘caladan’, ‘c’: ‘arrakis’ } )

>>> d1

{‘c’: ‘arrakis’, ‘b’: ‘caladan’, 5: ‘another integer’,

7: ‘an integer’ }
```

Python - Collections

Dictionaries

- A dictionary can be created from two lists:

```
mapping = {}

for key, value in zip( key_list, value_list ):

  mapping[key] = value
```

- Or:

```
>>> mapping = dict( zip( key_list, value_list ) )
```

Python - Collections
Dictionaries

- In order to be usable as a dictionary key, a Python object must be "hashable".
- Basic types in Python are hashable, but not mutable containers (e.g., lists).
- A hashable object implements `__hash__()`, `__eq__()`, and `__cmp__()` such that:
  1. The return value of `__hash__()` does not change during the life of the object.
  2. If two objects are equal according to `__eq__()` they must share the same `__hash__()` value.
  3. `__cmp__()` must compare objects consistently.

Python - Collections

Set

- A `set` is an unsorted collection of unique elements:

```
>>> set( [2,2,2,1,3,3] )

set([1, 2, 3])

>>> {2, 2, 2, 1, 3, 3}

set([1, 2, 3])
```

Python - Collections Set

`Method`	Alternate syntax	Description
`a.add(x)`	`--`	Add `x` to set.
`a.remove(x)`	`--`	Remove `x` from set.
`a.union(b)`	`a | b`	Union of `a` and `b`.
`a.intersection(b)`	`a & b`	Intersection of `a` and `b`.
`a.difference(b)`	`a - b`	Set difference.
`a.symmetric_difference(b)`	`a ^ b`	Symmetric set difference.
`a.issubset(b)`	`--`	`True` if `b` is a subset of `a`.
`a.issuperset(b)`	`--`	`True` if `a` is a superset of `b`.
`a.isdisjoint(b)`	`--`	`True` if `a` and `b` are disjoint.

Python - Collections

Comprehensions of lists, sets, and dictionaries

- Comprehensions are “syntactic sugar” to generate new collections by operating and filtering preexisting ones.
- The basic syntax is as follows:

```
>>> [expr for val in collection if condition]
```

equals:

```
new_list = []

for val in collection:

    if condition:

        new_list.append( expr )
```

Python - Collections

Comprehensions of lists, sets, and dictionaries

- The filtering condition may be omitted:

```
>>> strings = [‘a’, ‘an’, ‘the, ‘cat’, ‘car’, ‘pigeon’]

>>> [x.upper() for x in strings if len(x) > 2]

[‘THE’, ‘CAT’, ‘CAR’, ‘PIGEON’]

>>> [x.upper() for x in strings]

[‘A’, ‘AN’, ‘THE’, ‘CAT’, ‘CAR’, ‘PIGEON’]
```

Python - Collections

Comprehensions of lists, sets, and dictionaries

- Using a similar syntax we can write comprehensions of sets and dictionaries:

```
{ key-expr: val-expr for value in collection if condition }
```

```
{ set-expr for value in collection if condition }
```

Python - Collections

Comprehensions of lists, sets, and dictionaries

- We can write nested loops in a comprehension:

```
>>> tuples = ((1, 2, 3), (4, 5, 6), (7, 8, 9))
>>> [x for tup in tuples for x in tup if x > 3]
[4, 5, 6, 7, 8, 9]
```

- The order of the loops in a comprehension is the same as in an equivalent code:

```
>>> list = []
>>> for tup in tuples:
...     for x in tup:
...         if x > 3: list.append( x )
```

Python - Collections

Comprehensions of lists, sets, and dictionaries

- It is also valid to nest comprehensions:

```
>>> [[x for x in tup] for tup in tuples]

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Python - Functions

- Functions are defined using the reserved word `def`.
- `return` is used to return control to the caller and pass the result.

```
def my_func( x, y, z = 1.5 ):
    if z > 1:
        return z * (x+y)
    else:
        return z / (x+y)
```

Python - Functions

- If the end of the function code is reached without executing any `return` instruction, `None` is returned automatically.
- Functions receive two types of parameters: positional and keyword.
- Keyword arguments are commonly used to provide default values to optional parameters.
- In the example, `x` and `y` are positional parameters, while `z` is a keyword parameter. The function can be called in different ways:

```
>>> my_func( 5, 6,
z=0.7 )
6.14
```

```
>>> my_func( 3.14, 7,
3.5 )
17.990000000000002
```

Python - Functions

Name spaces, scopes, and local functions

- A function can access variables in two different scopes: global and local.
- Alternatively, variables can be explicitly defined inside a namespace.
- By default, variables assigned inside a function belong to the local scope.
- The local scope of a function is created when it is called, and initially contains its parameters.
- The local scope is destroyed when the function returns (except for closures, which we will briefly cover later).

Python - Functions

Name spaces, scopes, and local functions

```
def func():
    a = []
    for i in range(5):
        a.append( i )
```

- When `func()` is called, `a` is created. Then, the loop is executed and 5 integers are appended to `a`. Finally, the end of the function body is reached and `a` is destroyed.

Python - Functions

Name spaces, scopes, and local functions

```
>>> a = []
>>> def func():
...     for i in range(5):
...         a.append( i )
>>> func()
>>> a
[0, 1, 2, 3, 4]
```

Python - Functions

Name spaces, scopes, and local functions

```
>>> a = []
>>> def func():
...     a = range(5)
>>> func()
>>> a
[]
```

Python - Functions

Name spaces, scopes, and local functions

```
>>> a = []
>>> def func():
...     global a
...     a = range(5)
>>> func()
>>> a
[0, 1, 2, 3, 4]
```

Python - Functions

Name spaces, scopes, and local functions

- New functions can be declared anywhere in the code.
- In particular, it is legal to declare a function nested inside another function. These are called local functions, and are created when the enclosing function is called:

```
def outer_f( x, y, z ):
    def inner_f( a, b, c ):
        pass
    pass
```

- `inner_f()` does not exist until the call to `outer_f()`. As soon as `outer_f()` returns, `inner_f()` is destroyed.
- `inner_f()` can access the variables and functions in the local scope of `outer_f()`, but it cannot add anything to it.

Python - Functions
Returning multiple values

- A notable difference w.r.t. other languages such as C/C++/Java is the ability of a function to return multiple values.

```
def f():
    a = 5; b = 6; c = 7;
    return a, b, c
x, y, z = f()
```

- The implementation is very simple: `f()` is actually returning a single value, which happens to be a tuple.

```
>>> f()
(5, 6, 7)
```

Python - Functions
Functions as objects

- A Python function is just a special type of object which defines the `()` operator.
- As such, it is possible to use functors (pointers to function objects) to code complex operations in a simple way.
- E.g., we want to perform cleanup operations on the strings in the following array:

```
>>> planets = [ ‘    Caladan ’, ‘Ix!’, ‘Ix’, ‘ix’,
‘aRraKIs’, ‘giedi  prime##’, ‘Salusa secundus?’ ]
```

- We want to build a list of uniform strings for its analysis. We need to apply removal of unnecessary spaces and symbols, and to fix capitalization.

Python - Functions
Functions as objects

```
import re # Regular Expression module

def clean_strings( strings ):
    result = []
    for value in strings:
        value = value.strip()
        value = re.sub(‘[!#?]’, ‘’, value) # removes !#? symbols
        value = value.title()
        result.append( value )
    return result
```

Python - Functions
Functions as objects

```
def clean_strings_v2( strings, ops ):
    result = []
    for value in strings:
        for function in ops:
            value = function( value )
        result.append( value )
    return result

def remove_punctuation( value ):
    return re.sub( “[!#?]”, “”, value )

clean_ops = [ str.strip, remove_punctuation, str.title ]
```

Python - Functions
Functions as objects

- It is possible to pass functions as arguments to other functions.
- E.g., Python provides mechanisms to apply a function to a list of objects:

```
>>> map( str.strip, planets )

[‘Caladan’, ‘Ix!’, ‘Ix’, ‘ix’, ‘aRraKIs’, ‘giedi prime##’,
‘Salusa secundus?’]
```

Python - Functions
Anonymous (λ) functions

- An anonymous function, or lambda function, is a single-instruction functional expression the result of which is its return value:

```
def short_function(x):
    return x*2

equiv_anon = lambda x: x*2
```

Python - Functions
Anonymous (λ) functions

- Oftentimes it is simpler to use a reference to a lambda function than to write an ad-hoc, named one:

```
>>> a = [4, 0, 1, 5, 6]
>>> map( lambda x: x*2, a )
[8, 0, 2, 10, 12]
```

```
>>> strings = [‘caladan’, ‘ix’, ‘corrin’, ‘giedi prime’]
>>> strings.sort( key = lambda x: len(set(list(x))) )
# Sort by number of different letters in the string
>>> strings
[‘ix’, ‘caladan’, ‘corrin’, ‘giedi prime’]
```

Python - Functions
Closures

- A closure is a dynamically-generated function which is returned by another function.

- The distinguishing characteristic of a closure is that it is capable of accessing the local scope of its generating function after the latter returns.

Python - Functions
Closures

```
>>> def make_closure( a ):
...     def closure():
...         print( “Variable in the local scope: %d” % a )
...     return closure

>>> closure = make_closure(5)
>>> closure()
‘Variable in the local scope: 5’
```

Python - Functions
Closures

- In the example, a closure was created with an immutable internal state (the integer `a`).

- Mutable variables can also be used in a closure. These can be modified, dynamically altering the behavior of the closure.

Python - Functions

Closures

```
>>> def make_watcher():
...     have_seen = set([])
...     def has_been_seen( x ):
...         if x in have_seen: return True
...         else: have_seen.add(x)
...         return False
...     return has_been_seen
>>> watcher = make_watcher()
>>> vals = [5, 6, 1, 5, 1, 6, 3, 5]
>>> [watcher(x) for x in vals]
[False, False, False, True, True, True, False, True]
```

Python - Functions
Closures

- The local variables of a closure can be **modified**.

- No new variables can be **added** to the scope of a closure. A workaround is to add key/value pairs to a dictionary in the scope.

- Closures allow to build generic functions with plenty of options, that can be dynamically instantiated into specialized, efficient and simple variants.

Python - Functions
Closures

```
>>> def format_and_pad( template, space ):
...     def formatter( x ):
...         return (template % x).rjust( space )
...     return formatter
>>> fmt = format_and_pad( “%.4f”, 15 )
>>> fmt(1.756)
'         1.7560'
```

Python - Functions

Extended syntax: *args, **kwargs

- Functions are called using a mix of positional and keyword parameters:

```
>>> func( a, b, c, d = d_value, e = e_value )
```

- Internally, this function:
  1. Receives an `args` tuple containing its positional parameters.
  2. Receives a `kwargs` dictionary containing its keyword parameters:
  3. Performs the following assignment:

```
>>> (a, b, c) = args
>>> d = kwargs.get( ‘d’, d_default_value )
>>> e = kwargs.get( ‘e’, e_default_value )
```

Python - Functions

Extended syntax: *args, **kwargs

```
def g( x, y, z=1 ): return (x+y) / z

def hello_world_then_call( f, *args, *kwargs ):

    print ‘args is’, args

    print ‘kwargs is’, kwargs

    print “Hello world! Now I’m going to call %s” % f

    return f( *args, *kwargs )

>>> hello_world_then_call( g, 1, 2, z=5 )

args is (1, 2)

kwargs is {‘z’: 5.0}

Hello world! Now I’m going to call <function g at 0x2dd5cf8>

0.6
```

Python - Functions

Partial function application

- Partial function application consists in creating new functions from preexisting ones by fixing some of their parameters:

```
>>> def add(x, y): return x+y

>>> add_5 = lambda y: add(5,y)
```

- Alternatively:

```
>>> from functools import partial

>>> add_5 = partial( add, 5 )
```

Python - Functions
Generators

- A generator is a function which returns a sequence of values in a lazy way, stopping its execution after each value in the sequence.

- Generators are useful to generate large, iterable sequences in a memory-efficient way (e.g., `range()` vs `xrange()`).

- Generators are declared as a function which returns a value using `yield` instead of `return`.

Python - Functions

Generators

```
>>> def squares( n = 10 ):

...     for i in xrange( 1, n+1 ):

...         print “Generating squares from 1 to
%d”%(n**2)

...         yield i ** 2

>>> gen = squares()

>>> gen

<generator object squares at 0x7fd9e3e796e0>
```

Python - Functions
Generators

- When the generator function is called, no code is executed.
- Each element must be explicitly requested:

```
>>> for x in gen:
...     print x
Generating squares from 1 to 100
1
Generating squares from 1 to 100
4
.
.
.
```

Python - Functions
Generators

```
def make_change( amount, coins=[1, 2, 5, 10, 20, 50], hand=[] ):
    if amount == 0: yield hand
    for coin in coins:
        if coin > amount or (len(hand) > 0 and hand[-1] < coin):
continue
        for result in make_change(amount-coin, coins=coins,
hand=hand+[coin]):
         yield result


>>> len(list(make_change(53)))
530
```

Python - Functions
Generator expressions

- A compact way to declare a simple generator is to use a generator expression:

```
>>> gen = ( x ** 2 for x in xrange(100) )

>>> gen

<generator object <genexpr> at 0x7fd9e3e798c0>
```

- This expression is equivalent to:

```
def gen():

    for x in xrange(100):

        yield x**2
```

Python
Files and Operating System

- To open a file, `open()` is called passing a relative or absolute path:

```
>>> f = open( ‘folder/file.txt’ )
```

- By default, the file is opened in read-only mode (`‘r’`). The file can be seen as a collection of lines, and iterated using a `for` loop:

```
>>> for line in f:
...       # do something with line
```

Python
Files and Operating System

Opening mode	Description
r	Read only.
w	Write only. Creates a new file, overwriting any previous one.
a	Concatenate to a file (created if it does not exist).
r+	Read-write.
b	Binary mode (usage example: ‘rb’).
U	Use universal end-of-line mode. Translates any end-of-line marker in the file to ‘\n’.

Python
Files and Operating System

- To write to a file we can use the methods `write()` or `writelines()`:

- E.g., to remove white lines from a file:

```
>>> f_in = open( path, ‘r’)

>>> f_out = open( ‘tmp.txt’, ‘w’ )

>>> f_out.writelines( [x for x in f_in if len(x) > 1]
)
```

Python
Files and Operating System

Method	Description
`read( [size] )`	Reads data from the file as a string. The optional argument `[size]` is the number of bytes to read. Without it, the entire file is read.
`readlines( [size] )`	Same as `read()`, but returns a list of strings (one per line in the file). Without `[size]` the entire file is read.
`write( str )`	Writes `str` to the file.
`writelines( str )`	Writes a list of strings to the file, one per line.

Python
Files and Operating System

Method	Description
`close()`	Closes the file.
`flush()`	Synchronizes the I/O buffer to disk.
`seek( pos )`	Moves the file pointer to `pos`.
`tell()`	Returns the current position of the file pointer.
`closed`	`True` if the file is closed.

NumPy

- NumPy (***Num**erical **Py**thon*) is a fundamental package that enables efficient array and vector processing.
- It provides:
  - An `ndarray` class, which represents a multidimensional array and provides *vectorized* arithmetic operations and *broadcasting* capabilities.
  - *Vectorized* operations are applied to whole arrays (without using Python loops).
  - Operations for lineal algebra, randomness, signal processing, ...
  - Tools for the effective integration of Python and C/C++/Fortran.

NumPy - `ndarray`

`002-Numpy.ipynb`

- n-dimensional array object.
- Allows to perform operations on large data blocks with scalar syntax.
- Unlike lists, a NumPy array is usually homogeneous: it holds objects of a specific, predefined type.
- An `ndarray` contains, among others, the following two properties:
  - `shape`: tuple containing the array dimensions.
  - `dtype`: instance of the `dtype` class which specifies element datatype.

NumPy - `ndarray` Creation

002-Numpy.ipynb

Function	Description
`array()`	Builds a new array from the input sequence (e.g., a list). Copies input data to the new array.
`asarray()`	Converts the input sequence to an array, but it does not copy data if the input is already an `ndarray`.
`arange()`	Same as `range()` but returning an `ndarray` instead of a list.

`ones()`
`ones_like()`
	Build an array containing all `1s` from a tuple specifying the desired dimensions, or copying `shape` and `dtype` from another array.

NumPy - `ndarray` Creation

002-Numpy.ipynb

Function	Description

`zeros()`
`zeros_like()`
	Build an array containing all `0s` from a tuple specifying the desired dimensions, or copying `shape` and `dtype` from another array.

`empty()`
`empty_like()`
	Build an array without initializing its data from a tuple specifying the desired dimensions, or copying `shape` and `dtype` from another array.

`eye(),`
`identity()`
	Build an array with `1s` in its main diagonal and `0s` elsewhere. `identity()` requires square dimensions.

NumPy - `ndarray` Datatypes

Type	Code	Description
`int8`, `uint8`	`i8, u8`	Signed/unsigned integer, 8 bits (1 byte).
`int16`, `uint16`	`i16, u16`	Signed/unsigned integer, 16 bits (2 bytes).
`int32`, `uint32`	`i32, u32`	Signed/unsigned integer, 32 bits (4 bytes).
`int64`, `uint64`	`i64, u64`	Signed/unsigned integer, 64 bits (8 bytes).

NumPy - ndarray Datatypes

Type	Code	Description
float16	f2	Floating point, half precision (2 bytes).
float32	f4 o f	Floating point, single precision (4 bytes). Compatible with a C float.
float64	f64 o d	Floating point, double precision (8 bytes). Compatible with a C double and with a Python float.
float128	f16 o g	Floating point, extended precision (16 bytes).

NumPy - `ndarray` Datatypes

Type	Code	Description
`complex64`	`c8`	Complex number represented as 2 x `float32`.
`complex128`	`c16`	Complex number represented as 2 x `float64`.
`complex256`	`c32`	Complex number represented as 2 x `float128`.

NumPy - `ndarray` Datatypes

Type	Code	Description
`bool`	`?`	Boolean type (`True` or `False`).
`object`	`O`	Python object (reference).
`string_`	`S_`	String type with fixed length (1 byte per character). E.g., `S10` represents a 10-character string.
`unicode_`	`U_`	Unicode type with fixed length (the number of bytes per character varies per platform). Similar to `string_`.

NumPy - `ndarray` Datatypes

002-NumPy.ipynb

- `ndarray` can be cast to a different datatype using `astype()`.
- The call to `astype()` always creates a new copy of the array memory, even if the data is cast to its current type.
- If the cast fails for any reason, a `TypeError` exception is raised.

NumPy - `ndarray`

Broadcasting

- When two arrays are operated, NumPy must decide what operation to perform in the first place.
- If both arrays have the same dimensionality, the operation is performed in an element-wise fashion.
- Otherwise, NumPy tries to *broadcast* (replicate) the “smallest” array to match it to the largest one.

NumPy - `ndarray`

Broadcasting

- The broadcasting process begins by the innermost dimension (the rightmost ones), and moves towards the outer ones (to the left).
- Two dimensions are compatible if:
    1. Both have the same number of elements, or
    2. One of them has a single element.
- If none of the previous conditions is met, the operation raises a `ValueError`.
- If the second condition is met, the array with a single element in that particular dimension is copied `n` times to match both.

NumPy - `ndarray`

Broadcasting

002-NumPy.ipynb

- The arrays are not required to have the same number of dimensions. Non-existing dimensions are assumed to have a single element.
- For example:

```
A  : 256 x 256 x 3  A  : 256 x 256 x 3
B  :             3  B  : 256 x   1 x 3
A*B: 256 x 256 x 3  A*B: 256 x 256 x 3
```

NumPy - `ndarray`

Basic indexing

`002-NumPy.ipynb`

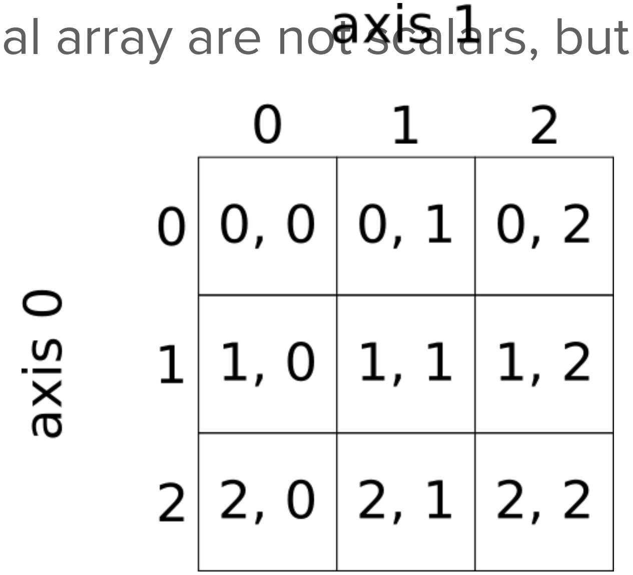
- 1-dimensional arrays behave in a similar way to lists.
- An important difference is that in NumPy a slice is not a copy, but a *view* over the array data. This improves performance, but also means that slice operations may have collateral effects on the original array.
- It is possible to obtain a copy of a slice (or any array) by calling the `copy()` method of `ndarray`.

NumPy - `ndarray`

Basic indexing

002-NumPy.ipynb

The elements in each index of an n-dimensional array are not scalars, but (n-1)-dimensional arrays.



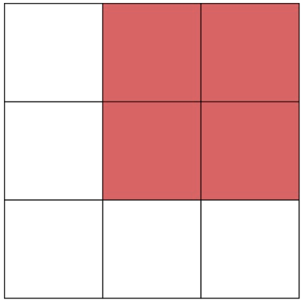
	0	1	2
0	0, 0	0, 1	0, 2
1	1, 0	1, 1	1, 2
2	2, 0	2, 1	2, 2

NumPy - `ndarray`

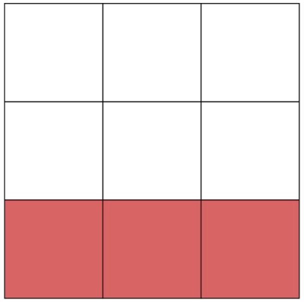
Indexing with slices

002-NumPy.ipynb

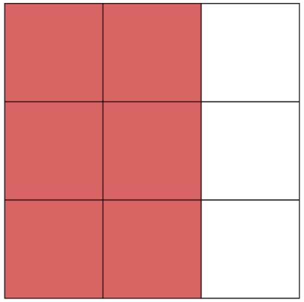
- The slice syntax is also valid for indexing arrays.
- Working with n-dimensional arrays, it is possible to use any combination of basic indexing and slicing.



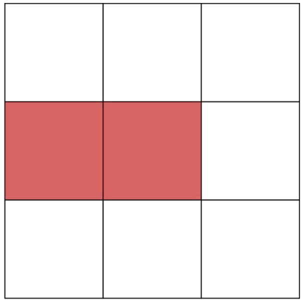
`arr[:2, 1:]`
`shape = (2,2)`



`arr[2]`
`shape = (3,)`



`arr[:,:2]`
`shape = (3,2)`



`arr[1:2,:2]`
`shape = (1,2)`

NumPy - `ndarray`

Boolean indexing

002-NumPy.ipynb

- It is possible to index an array using a boolean array.
- The boolean array must have the same dimensionality as the axis it must index.
- Boolean indexing can be used to select elements in an array using data semantics (e.g., the elements which are greater than a particular number). The conditions do not need to refer to the array being indexed.
- Boolean indexing always returns a copy of the data (not a view).
- When accessing an n-dimensional array, boolean indexing can be mixed together with basic and sliced indexing.

NumPy - `ndarray`

Fancy indexing

002-NumPy.ipynb

- The term *fancy indexing* describes the indexing of an array using an integer collection containing the indices of the elements to select.
- If a list of lists is provided, the behavior changes: a 1D array is returned, containing the data indexed by the tuples resulting from applying `zip()` to the input lists.
- Fancy indexing always returns a copy of the data, not a view.
- When accessing an n-dimensional array, fancy indexing can be mixed together with basic, sliced, and boolean indexing.

NumPy - `ndarray`

Transposition

002-NumPy.ipynb

- `ndarray` provides the `transpose()` method and the `T` attribute.
- For n-dimensional arrays, `transpose()` accepts as input a tuple specifying an arbitrary dimensional permutation.
- The T attribute is a quick way of accessing “clasic” transposition, in which a matrix is flipped over its main diagonal.
- The `swapaxes()` is used to swap any two axes.
- All these operations return a view over the original array.

NumPy - Universal functions

`002-NumPy.ipynb`

- A universal function, or `ufunc`, performs element-wise operations over an `ndarray`.
- The term describes a wrapper over a simple function which reads one or more scalars and returns a single scalar.
- Many `ufunc`s take a single element as input (e.g., `sqrt()` or `exp()`). These are *unary* `ufunc`s.
- Other `ufunc`s take two elements and return one (*binary* `ufunc`s, e.g., `add()` or `maximum()`).
- Some `ufunc`s return more than one array (e.g., `modf()` returns the integral and fractional parts of a floating point array).

NumPy - Universal functions
Unary `ufuncs`

Function	Operation
`abs, fabs`	Absolute value of integers, floating point or complex numbers. `fabs()` is a faster version for non-complex numbers.
`sqrt`	Square root: `arr ** 0.5`.
`square`	Square: `arr ** 2`.
`exp`	Exponentiation: `np.e ** arr`.
`log, log10, log2, log1p`	Natural, base 10, and base 2 logarithms. `log1p(x)` returns `log(1+x).`

NumPy - Universal functions
Unary `ufuncs`

Function	Return value
`sign`	The sign of each element in the array: 1 (positive), 0 (zero), -1 (negative).
`ceil / floor / rint`	Round up / down / to closer integer.
`modf`	Integral and fractional parts of an array.
`isnan`	`True` if an element is `np.nan` (NaN).
`isfinite, isinf`	Boolean array indicating if an element is finite / infinite.

NumPy - Universal functions
Unary `ufuncs`

Function	Return value
`cos, cosh, sin, sinh, tan, tanh`	Trigonometric and hyperbolic functions.
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric and hyperbolic functions.
`logical_not`	Element-wise boolean NOT.

NumPy - Universal functions

Binary `ufuncs`

Function	Return value
`add`	Addition of two input arrays.
`subtract`	Subtracts the second parameter from the first.
`multiply`	Multiplication of two input arrays.
`divide / floor_divide`	Division / integral division of the input.
`power`	Exponentiation of the bases in the first array to the exponents in the second array.

NumPy - Universal functions

Binary `ufuncs`

Function	Return value
`maximum / fmax`	Element-wise maximum. `fmax()` ignores NaN values.
`minimum / fmin`	Element-wise minimum. `fmin()` ignores NaN values.
`mod`	Element-wise modulo.
`copysign`	Copies the signs of the element in the second array to the values in the first.

NumPy - Universal functions
Binary `ufuncs`

Function	Return value
`greater / greater_equal / less / less_equal / equal / not_equal`	Boolean operations comparing wether each element in the first array is greater / greater or equal / less / less or equal / equal / different than each element in the second array.
`logical_and / logical_or / logical_xor`	Element-wise boolean operations AND / OR / XOR.

NumPy - Data processing

002-NumPy.ipynb

- Using NumPy we can express data processing algorithms that would otherwise require loops using array operations.

- This *vectorization* process usually comes together with significant performance improvements.

- We will explore different ways of using NumPy to process data.

NumPy - Data processing
Conditional logic

002-NumPy.ipynb

- `numpy.where()` allows to write a vectorized version of the expression `x if c else y`.
- Using pure Python, this can be written as:

```
>>> result = [(x if c else y) for x, y, c in zip(xarr, yarr, cond)]
```

- However, this version…
  - Is very slow.
  - works with 1-dimensional arrays only.
- The equivalent `where()` version is written:

```
>>> result = np.where( cond, xarr, yarr )
```

NumPy - Data processing
Mathematics and statistics

Method	Description
`sum`	Summation of all the elements of an array along a given optional axis.
`mean`	Arithmetic mean.
`std, var`	Standard deviation and variance.
`min, max`	Minimum and maximum.

NumPy - Data processing
Mathematics and statistics

002-NumPy.ipynb

Method	Description
`argmin, argmax`	Indices of minimum and maximum elements.
`cumsum`	Cumulative sum.
`cumprod`	Cumulative product.

NumPy - Data processing
Boolean arrays

002-NumPy.ipynb

- When working with boolean arrays, the previous methods use `True == 1` and `False == 0`. This allows to reduce boolean values using `cumsum()` / `cumprod()` instead of AND / OR.

- There are however ad-hoc methods for reducing boolean arrays: `any()` and `all()`.

- These methods work with non-boolean arrays by interpreting `non-zero == True`, `zero == False`.

NumPy - Data processing
Sorting

002-NumPy.ipynb

- As with lists, NumPy arrays can be sorted in place using `sort()`.

- `sort()` accepts an optional `axis` parameter which indicates the axis to sort over in multidimensional arrays.

- `numpy.sort()` returns a sorted copy of the array.

NumPy - Data processing
Set logic

Function	Description
`unique(x)`	Computes the unique elements in array `x` (1D).
`intersect1d(x,y)`	Common elements in `x` and `y`.
`union1d(x,y)`	Union of `x` and `y`.
`in1d(x,y)`	Boolean array indicating whether each element of `x` is in `y`.
`setdiff1d(x,y)`	Set difference, `x`-`y`.
`setxor1d(x,y)`	Symmetric set difference.

NumPy - Array I/O
Storing arrays to disk

- `np.save()` and `np.load()` store arrays to disk in binary format.
- By default, the data is stored uncompressed with extension `.npy`.

```
>>> np.save( path_string, arr )

.

.

.

>>> arr = np.load( path_string )
```

NumPy - Array I/O
Storing arrays to disk

- We can store several arrays to the same file compressed using ZIP and with `.npz` extension using `np.savez()`:

```
>>> np.savez( path_string, a = arr1, b = arr2 )
```

- Reading a `.npz` file NumPy returns a dictionary object that loads individual arrays in a lazy way:

```
>>> arch = np.load( path_string )

>>> arch[‘b’]

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

NumPy - Array I/O
Storing arrays to disk

- It is often useful to read arrays stored in text format.
- NumPy provides `np.loadtxt()` and `np.genfromtxt()`.
- Both can be adapted to a specific input format, varying comments format, delimiters, conversion functions, rows to be skipped, or columns to parse.
- `np.savetxt()` performs the reverse operation: it writes an array to a CSV-like file.
- `np.genfromtxt()` is similar to `np.loadtxt()`, but it supports treatment of missing values and structuring of the output array.

NumPy - Linear algebra (`numpy.linalg`)

Function	Description
`diag`	Returns the elements in the diagonal of a square matrix as a 1D array, or builds a square matrix from a 1D array.
`dot`	Matrix product.
`trace`	Sum of the elements in the matrix diagonal.
`det`	Determinant.
`eig`	Eigenvalues and eigenvectors.

NumPy - Linear algebra (`numpy.linalg`)

Function	Description
`inv`	Inverse of an square matrix.
`pinv`	Moore-Penrose pseudo-inverse.
`qr`	QR decomposition.
`svd`	Singular-value decomposition.
`solve`	Solves `Ax = b`, with `A` a square matrix.
`lstsq`	Least squares solution to `y = Xb`.

NumPy - Random numbers (`numpy.random`)

Function	Description
`seed`	Changes the generator seed.
`permutation`	Random permutation of a sequence.
`shuffle`	Random permutation of a sequence (in-place).
`rand`	Samples a uniform distribution.
`randint`	Samples integers from a uniform distribution.

NumPy - Random numbers (`numpy.random`)

Function	Description
`randn`	Samples a standard normal distribution.
`binomial`	Samples a binomial distribution.
`normal`	Samples a normal distribution.
`beta`	Samples a beta distribution.

NumPy - Random numbers (`numpy.random`)

002-NumPy.ipynb

Function	Description
`chisquare`	Samples a χ^2 distribution.
`gamma`	Samples a gamma distribution.
`uniform`	Samples a uniform `[0,1)` distribution.

Matplotlib

- The previous examples included some plotting, but without going into details.
- Matplotlib provides mostly 2D plotting capabilities, although it includes limited 3D functionalities.
- The basic supported plot types are lines, bar charts, histograms, pie charts, and variations of them.
- This section briefly introduces Matplotlib. It has a vast amount of options that cannot be covered in detail. We will later focus on higher level libraries instead.

Matplotlib - Introduction

- The `%matplotlib` magic command automatically configures iPython to show plots.
- By default, iPython detects the proper backend for the current window manager. Using `%matplotlib inline` the plots will be shown embedded inside a QT or Notebook environment.
- A basic Matplotlib plot includes the following elements:
    - `x` and `y` axes: horizontal and vertical axes, respectively.
    - Tick marks in the `x` and `y` axes.
    - Tick labels, showing axis values.
    - Drawing area, called canvas.

Matplotlib - Introduction

003-Matplotlib.ipynb

- `plot()` is used to draw lines and marks.
- It accepts pairs of `x` and `y` sequences, which must have the same lengths, together with a string indicating how to plot the data.
- Adjacent points are joined using straight lines.
- Points and lines are drawn following the requested style.
- The function returns a list of the lines which have been added to the current figure.
- If only one sequence is passed to the function, it is assumed to contain the values of the `y` axis. Values for the horizontal axis will be automatically generated as `x = range( len(y) ).`

Matplotlib - Introduction

`003-Matplotlib.ipynb`

Function	Description
`figure`	Creates a new figure. Accepts an integer that acts as a unique identifier for this figure. This integer can be used to programatically change the active figure.
`subplot( x, y, z )`	Divides a figure into a mesh of subfigures with `x` rows and `y` columns. Besides, it activates subfigure number `z` (`1 < z < x*y`, row major order).

Matplotlib - Introduction

003-Matplotlib.ipynb

Function	Description
`bar / barh`	Create vertical / horizontal bar charts. To plot stacked bars the `bottom` parameters is used, indicating the starting point for the new bars.
`boxplot`	Box plots.
`scatter`	Draws points, but it does not join them with lines unlike `plot()`.
`hist`	Histograms.
`pie`	Pie charts.

Matplotlib - Design

`003-Matplotlib.ipynb`

Function	Description
`title`	Plot title. Like all other text-related Matplotlib functions, it accepts LaTeX syntax.
`xlim / ylim`	Configures the limits of the `x` / `y` axes.
`autoscale`	Automatic axes limits.
`xticks / yticks`	Configures the ticks for each axis and, optionally, the tick labels.

Matplotlib - Design

003-Matplotlib.ipynb

Function	Description
`axis`	Configures both axes’ limits.
`axhline / axvline`	Draws a horizontal / vertical line,
`axhspan / axvspan`	Draws rectangles that cover the entire width / height of the plot.

Matplotlib - Design Axes

003-Matplotlib.ipynb

- A reference to an instance of `matplotlib.axes.Axes` allows to configure plots with a high level of detail.
- Such reference can be obtained by calling `matplotlib.pyplot.gca()` (Get Current Axes).
- E.g., calling `ax.xaxis.set_major_locator( matplotlib.ticker.MultipleLocator(10))` sets ticks in the `x` axis of the `ax` axes in values multiple of 10.
- E.g., calling `ax.xaxis.set_major_formatter()` allows to specify functions that will be used to format the tick labels in the `x` axis of the `ax` axes.

Matplotlib - Design
Legends and annotations

003-Matplotlib.ipynb

- All plotting functions (`plot()`, `hist()`, etc.) accept a `label` parameter indicating the name to use in the legend.
- `matplotlib.pyplot.legend()` automatically builds a legend from a list of handles. If no such list is provided, all the elements in the figure with a label not starting with “_” are listed.
- Otherwise, we can provide the lists of lines, etc. returned by each plotting function to specify which elements in the figure should be added to the legend.
- `matplotlib.pyplot.annotate()` adds a textual annotation and the specified place in the figure.

Matplotlib - Styles

003-Matplotlib.ipynb

Property	Value type	Description
`alpha`	`float`	Transparency used for the element.
`color`	Matplotlib color	Line / marker color.
`dashes`	Sequence	Line pattern.
`label`	`string`	Text to label this element in the plot legend.
`linestyle`	See docs	Line type.
`linewidth`	`float`	Line width, in points.

Matplotlib - Styles

003-Matplotlib.ipynb

Property	Value type	Description
marker	See docs	Marker used for points in a line.
mec	Matplotlib color	Marker Edge Color.
mew	float	Marker Edge Width.
mfc	Matplotlib color	Marker Face Color.
markersize	float	Marker size, in points.

Matplotlib - Styles

003-Matplotlib.ipynb

Property	Value type	Description
solid_capstyle	[‘butt’|‘round’|‘projecting’]	End of line style.
solid_joinstyle	[‘miter’|‘round’|‘bevel’]	Line union style.
visible	[True | False]	Visibility.
xdata	numpy.ndarray	X axis data.
ydata	numpy.ndarray	Y axis data.
Zorder	Número	Stacking order in the Z axis.

Matplotlib - Colors

003-Matplotlib.ipynb

- The `matplotlib.colors` module includes utilities for defining and converting colors.
- Basic predefined colors can be referenced using a single letter: b (*blue*), g (*green*), r (*red*), c (*cyan*), m (*magenta*), y (*yellow*), k (*black*), w (*white*).
- Shades of gray can be codified using a floating point number in `[0, 1]`.
- Other colors can be specified using different formats:
  - HTML hex string: `“#eeefff”`
  - `(R, G, B)` tuple, with `R`, `G` and `B` in `[0, 1]`.
  - HTML string: `“red”`, `“burlywood”`, `“chartreuse”`, ...

Matplotlib - Colors
Colormaps

003-Matplotlib.ipynb

- The `matplotlib.cm` module includes a set of colormaps to use with plots.

- Colormaps are useful when using `imshow()`, which is similar to `plot()` but interprets the values in the input array as indices to a colormap. If the input array is 2-dimensional, an image will be plotted.

Matplotlib - Saving to file

003-Matplotlib.ipynb

- Figures can be saved to a file using `matplotlib.pyplot.savefig()`.
- The file format to use is inferred from the extension in the provided path.
- The most relevant parameters that control the quality of the stored figure are:
  - `dpi`: dots per inch.
  - `bbox_inches`: inches of whitespace surrounding the figure.
- The figure does not need to be stored to a file: it can be written to any object which supports I/O, such as `StringIO`.

Matplotlib - 2D histograms

Parameter to `hist()`	Description
`bins`	Number of classes to use in the histogram, or sequence containing the boundaries between classes.
`range`	Range for each class. If `bins` is not used, this can be provided as a sequence.
`normed`	If `True`, values are normalized and the result is a probability density function.

Matplotlib - 2D histograms

003-Matplotlib.ipynb

Parameter to **`hist()`**	**Description**
`histtype`	By default, bar chart. Other values: ● `barstacked`: stacks bars when working with multiple data series. ● `step`: line without filling. ● `stepfilled`: line with filling.
`align`	How to align bars for each class: `mid`, `left`, o `right`.
`color`	Colors to be used.
`orientation`	`horizontal` or `vertical`.

Matplotlib - Pie charts

003-Matplotlib.ipynb

Parameter to `pie()`	Description
`explode`	Fraction of the pie ratio to use as offset for each slice.
`autopct`	String or function indicating how to label each slice with a numeric value.
`pctdistance`	Ratio between the center of each slice and the start of the `autopct` text.
`labeldistance`	Radial distance to draw labels.
`startangle`	Rotation angle of the origin.
`wedgeprops`	Dictionary containing slice properties.

Matplotlib - 3D plots

003-Matplotlib.ipynb

- Although Matplotlib is focused in 2D plots, there are several toolkits provided limited 3D capabilities.

- The `mpl_toolkits.mplot3d` module provides methods to create 3D scatter plots, surfaces, lines and meshes. Its interface is very similar to the original Matplotlib.

- Main difference: axes are instances of `mpl_toolkits.mplot3d.Axes3D`. Projections are performed by specialized 3D classes, while other parts of the figure (labels, ticks, etc.) are managed by vanilla Matplotlib.

Bibliography

- *Learning Python*. Mark Lutz. O’Reilly Media, 2013 (5ª edición).
- *Programming Python.* Mark Lutz. O’Reilly Media, 2011 (4ª edición).

Data Analytics: Pandas

- Intro to Pandas.
- Data I/O.
- Data wrangling.
- Visualization.
- Aggregation.
- Time series.

Pandas

- Pandas provides data structures and methods to improve the structured data processing capabilities of native Python.
- The basic data structure in Pandas is the `DataFrame` (similar to `data.frame` in R). It is a 2D table, conceptually similar to an Excel spreadsheet.
- Pandas combines the array processing capabilities of NumPy with the flexibility of spreadsheets and relational databases.
- Provides indexing, reshaping, splitting, aggregation, and selection of subsets of data.

Pandas

- Pandas design objective is to provide new data management capabilities to the Python ecosystem:
  - Data structures with indexed axes supporting explicit or implicit data alignment.
  - Seamless processing of time series (timestamp-indexed data).
  - Arithmetic operations and reductions along axes.
  - Flexible management of unknown (null) data.
  - Merges and similar operations typical of relational databases.

Pandas - Data structures

005-Pandas-Intro.ipynb

Series

- A Series object represents an object conceptually similar to a 1D array.

- It actually contains two different arrays: a data array and an index array, which labels data.

- Both arrays are esentially NumPy arrays with their own datatypes.

- Series are similar to dictionaries, as they can be seen as key-value pairs. In fact, a constructor is provided to build a Series object from a dict.

Pandas - Data structures

`005-Pandas-Intro.ipynb`

`DataFrame`

- A `DataFrame` object represents a tabular structure, similar to a spreadsheet (or a `data.frame` in R).
- It contains an ordered collection of colums, each of which may have a different datatype (numeric, string, etc.).
- Includes an index on its rows and another one on its columns. In this sense, it can be seen as an aggregation of series, all of them sharing the same index.
- Internally, the data is stored in a 2D format (bidimensional NumPy array), although higher dimensional data may be represented using hierarchical indices.

Pandas - Data structures

`DataFrame()` constructor

Parameter (type)	**Description**
`2D ndarray`	Data array, with optional row and column labels.
`dict of arrays, lists, or tuples`	Each sequence becomes a column of the `DataFrame`. All of them might have the same length.
`dict of Series`	Each value becomes a column. The indices in each series are unified if an explicit index is not provided.

Pandas - Data structures

`DataFrame()` constructor

Parameter (type)	Description
`dict of dicts`	Each internal dictionary becomes a column. The keys in the different dictionaries are unified as in a `dict` of `Series`.
`list of dicts or Series`	Each item becomes a row in the `DataFrame`. The union of the keys of the dictionaries or the indices of the series is used for the column labels.
`list of lists or tuples`	Same as providing an `2D ndarray.`

Pandas - Data structures

`DataFrame()` constructor

Parameter (type)	Description
`DataFrame`	The indices already in the old DataFrame are used, unless different ones are explicitly provided.

Pandas - Data structures

005-Pandas-Intro.ipynb

`Index`

- `Index` objects are responsible for storing axis labels and names.

- Any other array or sequence type provided as an index when building a `Series` or `DataFrame` object is internally converted to an `Index`.

- `Index` objects are immutable. This guarantees referential integrity when shared by different structures.

- `Index` provides methods and attributes to support set logic and value inspection.

Pandas - Data structures

`Index` subclasses

Class	Description
`Index`	Array of generic Python objects.
`Int64Index`	Integers.
`MultiIndex`	Hierarchical index, representing multiple indexing levels in a single axis. Similar to a tuple array.
`DatetimeIndex`	Timestamp with nanosecond resolution.
`PeriodIndex`	Time periods.

Pandas - Data structures

`Index` methods

Method	Description
`append`	Concatenates additional `Index` objects, producing a new object.
`diff`	Set difference.
`intersection`	Set intersection.
`union`	Set union.
`isin`	Computes a boolean array marking whether each of the entries in an `Index` is included in another collection.
`delete`	Creates a new `Index` object by removing an element from the original one.

Pandas - Data structures

Index methods

Method	Description
drop	Creates a new Index object by removing a set of elements from the original one.
insert	Creates a new Index object by inserting a new element into the original one.
is_monotonic	True if each element is greater or equal than the previous one. Alias for is_monotonic_increasing(). An is_monotonic_decreasing() method is also provided.
is_unique	True if the index has no duplicate elements.
unique	Computes an array containing the unique elements in the index.

Pandas - Essential functions

005-Pandas-Intro.ipynb

- Reindexing: `reindex()` method.
- Removing entries: `drop()` method.
- Indexing, selection, and filtering: operator `[]` and attributes `DataFrame.loc` and `DataFrame.iloc`.
- Arithmetic operations: operators `+`, `–`, `*`, and `/`, and methods `add()`, `sub()`, `mul()` y `div()`.
- Functional application and mapping: basic methods (`mean()`, `sum()`, ...), method `Series.map()`, methods `DataFrame.apply()`, and `DataFrame.applymap()`.
- Sorting and classification: methods `sort_values()`, `sort_index()`, and `rank()`.
- Managing indices with duplicates: method `Index.is_unique()`.

Pandas - Essential functions

`reindex()` parameters

Parameter	Description
`index`	New sequence to use as index.
`method`	Interpolation method (`ffill` or `bfill`).
`fill_value`	Value to use as a placeholder for null data.
`limit`	Maximum number of elements to fill using interpolation.
`level`	Hierarchical level to reindex.
`copy`	Marks whether data should be copied in case the new and the old indices are equivalent (`True` by default).

Pandas - Essential functions

`DataFrame` indexing

Syntax	Description
`obj[val]`	Selects a column or subset of columns, except if `val` is an array or the `DataFrame` is boolean, in which case it filters colums.
`obj.loc[val]`	Selects a row or subset of rows by label.
`obj.loc[:,val]`	Selects a column or subset of columns, by label.
`obj.loc[val1, val2]`	Selects both rows and columns, by label.

Pandas - Essential functions

`DataFrame` indexing

Syntax	**Description**
`reindex()`	Reorganizes one or more axes according to new indices.
`xs()`	Returns a cross-section of the dataframe attending to its labels.
`icol() / irow()`	Selects a single row / column attending to its location.
`get_value() / set_value()`	Selects a single value attending to row/column labels.

Pandas - Descriptive statistics
Reduction parameters

Parameter	Description
`axis`	The axis on which to perform the reduction (rows=`0`, colums=`1`).
`skipna`	Whether to exclude null values. `True` by default.
`level`	Reduction grouping by `level` in hierarchical indices.

Pandas - Descriptive statistics Methods

Method	Description
`count`	Number of non-null values.
`describe`	Computes several statistics of a `Series` or the columns of a `DataFrame`.
`min / max`	Minimum / maximum value.
`argmin / argmax`	Location of the minimum / maximum value.
`idxmin / idxmax`	Index (label) of the minimum / maximum value.
`quantile`	Returns the specified p-quantile.
`sum`	Summation.

Pandas - Descriptive statistics Methods

Method	Description
`mean`	Arithmetic mean.
`median`	Median.
`mad`	Mean absolute deviation.
`var`	Variance.
`std`	Standard deviation.
`skew`	Skewness (third standardized moment).
`kurt`	Kurtosis (fourth standardized moment).

Pandas - Descriptive statistics Methods

Method	Description
`cumsum`	Cumulative sum.
`cummin, cummax`	Cumulative minimum / maximum.
`cumprod`	Cumulative product.
`diff`	First order differences.
`pct_change`	Percentage change.

Pandas - Unknown data

005-Pandas-Intro.ipynb

- Unknown or null data are common in most data analysis applications.

- Pandas simplifies the management of null data, e.g., all descriptive statistics functions automatically omit unknown values.

- `numpy.nan` is used as the default placeholder for unknown data. `None` will also be treated as a null value by Pandas.

Pandas - Unknown data
fillna() parameters

Parameter	Description
value	Escalar value or dictionary to use for filling.
method	Interpolation type (“ffill” or “bfill”).
axis	Axis to fill (by default 0, i.e., rows).
inplace	Modifies the object inplace, instead of creating a new copy.
limit	Maximum number of consecutive values to fill using interpolation.

Pandas - Hierarchical indices

005-Pandas-Intro.ipynb

- Hierarchical indexing allows to works (at a conceptual level) using `DataFrame` objects with more than 2 dimensions.
- It is implemented through the addition of different *levels* to the row and/or column indices of the table.
- Hierarchical indices are implemented by the `MultiIndex` class.
- It is possible to apply reduction operations to the different levels in the index hierarchy, obtaining a `DataFrame` (instead of a `Series`) after the reduction.

Pandas - Other considerations
Integer indexing

- Pandas objects with integer indices can be confusing, as the semantics of position- and label-based indexation vary.

```
obj = Series( range(5), index=[‘a’, ‘b’, ‘c’, ‘d’, ‘e’] )

obj[-1]

??



obj = Series( range(5), index=range(5) )

obj[-1]

??
```

Pandas - Other considerations
Integer indexing

- For objects with an integer index, Pandas cannot decide whether the user wants to apply location- or label-based indexing.
- It solves the ambiguity by always using label-based indexing.
- Provides methods `Series.iget_value()` and `DataFrame.irow()` for positional-based indexing in integer-indexed data.

Pandas: Data I/O
Reading / writing text

006-Pandas-DataIO.ipynb

- Pandas provides a set of methods to create `DataFrame` objects from tabular data stored in text format. The most useful ones are `read_csv()` and `read_table()` (which are nowadays mostly equivalent), which include options for:
  - Using one or more columns in the text file as `DataFrame` indices.
  - Naming columns using parameters or extracting names from the file.
  - Inferring types and performing data conversions.
  - Parsing dates, including combining several colums into a single one.
  - Iterating over chunks of large files (to fit data into memory).
  - Cleaning data: ignoring some rows or columns, comments, etc.

- Automatic type inference implies that it is not required to specify column types. Management of data and other non-basic types takes extra effort.

Pandas: Data I/O

`read_csv` / `read_table` parameters

Parameter	Description
`path`	URL of the file to open.
`sep / delimiter`	Regular expression to use for separating fields.
`header`	Row number containing the colum names (`0` by default), or `None` if correlative integers should be used.
`index_col`	Columns to build the `DataFrame` index.
`names`	Names for the `DataFrame` columns. Combine with `header=None`.
`skiprows`	Number of rows to ignore at the beginning of the file, or list of row numbers to ignore.

Pandas: Data I/O

`read_csv` / `read_table` parameters

Parameter	Description
`na_values`	Values that should be considered marks of unknown data.
`comment`	Regular expression to mark the beginning of a comment.
`parse_dates`	Tries to parse dates to a `datetime` object. `False` by default. If `True`, Pandas tries to parse all columns as dates. Alternatively, it can be a list of specific columns to parse as dates. If an element in the list is a tuple/list, it will try to combine the specified columns to parse a single date.
`keep_date_col`	If several columns are used to build a single date column, do not keep the joined columns. `True` by default.

Pandas: Data I/O

`read_csv` / `read_table` parameters

Parameter	Description
`converters`	Dictionary containing a mapping of columns to parsing functions. Each corresponding function will be applied to all the elements of a given column and the result will be inserted into the `DataFrame`.
`dayfirst`	When parsing potentially ambiguous dates, assume dates in international format (DD/MM/YYYY). `False` by default.
`date_parser`	Function to use to parse dates.
`nrows`	Number of rows to read, starting from the beginning of the file.
`iterator`	If `True`, the return value will be a `TextFileReader` object, to process the file in a chunk-by-chunk manner.

Pandas: Data I/O

`read_csv` / `read_table` parameters

Parameter	Description
`chunksize`	If chunk-by-chunk processing is active, size of each chunk.
`skip_footer`	Number of rows to ignore at the end of the file.
`verbose`	If `True`, print information about the parsing process.
`encoding`	Character encoding, e.g., ‘utf-8’.
`squeeze`	If `True`, in case the result contains a single column, return a `Series`.
`thousands`	Thousands separator, e.g., ‘,’ or ‘.’

Pandas: Data I/O
Binary formats

`006-Pandas-DataIO.ipynb`

- The `pickle` module in the Python standard library provides a convenient method to serialize (marshall) and deserialize (unmarshall) objects to binary format.
- Pandas provides the methods `to_pickle()/read_pickle()` which store/read data to/from `pickle` files.
- Using `pickle` is discouraged for long term storage, since `pickle` does not guarantee backwards compatibility.
- Pandas also provides methods to read and write Excel, HDF5, Stata, and HTML files, among others.

Pandas: Data I/O
HTML and web services

006-Pandas-DataIO.ipynb

- Many web sites include APIs that provide access to data sources in JSON format.
- There are several ways to access these services from Python.
- A simple way is through the `requests` package.
- The responses to requests will be converted to a JSON object.
- It is trivial to build a Pandas object from JSON, as we have seen.

Pandas: Data I/O
Databases

006-Pandas-DataIO.ipynb

- Text and binary files are ultimately inefficient to store large amounts of data.
- Databases, both relational and non-relational, are one of the most common data sources in computer science.
- Pandas provides methods to load data from SQL queries. The `pandas.io.sql` module allows to execute SQL sentences and process the results.
- Other types of non-SQL databases, such as MongoDB, store objects in different formats: JSON, text, etc. Loading data mechanisms vary on a per-case basis.

Pandas: Data wrangling

- Much of the programming work in data analysis is spent on data preparation: loading, cleaning, transforming, and rearranging.
- Pandas and Python provide a set of flexible, high level data manipulation tools suitable for these preparation tasks:
  - Merging and combination of data: `merge()`, `concat()`, `combine_first()`.
  - Reshaping and pivoting: `stack()`, `unstack()`, `pivot()`.
  - Data transformation: removing duplicates, functional application, replacing values, renaming axes, discretization and binning, detecting and filtering outliers, permutation, and random sampling.
  - String manipulation: normalizing, cleaning, use of regular expressions...

Pandas: Data wrangling
Combining and merging datasets

`007-Pandas-Data-wrangling.ipynb`

- `pandas.merge()` connects rows in `DataFrame` based on one or more keys. It is similar to a `join` operation in a relational database.

- `pandas.concat()` glues or stacks together objects along an axis:
    - How are the new axes labeled? (union, intersection, ...)
    - Are the original groups identifiable in the resulting object?
    - Which axis should be used for concatenating?

- `combine_first()` enables splicing together overlapping data to fill in missing values in one object with values from the other.

Pandas: Data wrangling

`merge()` parameters

Parameter	Description
`left`	Left hand side operand of the merge.
`right`	Right hand side operand of the merge.
`how`	`[‘inner’|’outer’|’left’|’right’].` `‘inner’` by default.
`on`	Column names to join on. Must be found in both `DataFrame` objects. If not specified, will use all columns with matching names.

Pandas: Data wrangling

`merge()` parameters

Parameter	Description
`left_on`	Columns in *left* operand to use as join keys.
`right_on`	Columns in *right* operand to use as join keys.
`left_index`	`[True|False]`. Use row index in *left* as its join key.
`right_index`	`[True|False]`. Use row index in *right* as its join key.

Pandas: Data wrangling

`merge()` parameters

Parameter	Description
`sort`	`[True|False]`. Sort merged data lexicographically by join keys. `True` by default. Disable to get better performance in some cases on large datasets.
`suffixes`	Tuple of string values to append to column names in case of overlap. `(‘_x’, ‘_y’)` by default.
`copy`	If `False`, avoid copying data into resulting data structure in some exceptional cases. By default always copies.

Pandas: Data wrangling

`concat()` parameters

Parameter	Description
`objs`	List or dictionary of Pandas objects to be concatenated. The only required argument.
`axis`	Axis to concatenate along. `0` by default (rows).
`join`	`[‘inner’|’outer’]`. `‘outer’` by default. Whether to intersect (`inner`) or union (`outer`) together indices along the other axes.
`join_axes`	Specific indices to use for the other `(n-1)` axes instead of performing union/intersection logic.

Pandas: Data wrangling

`concat()` parameters

Parameter	Description
`keys`	Values to associate with objects being concatenated, forming a hierarchical index along the concatenation axis.
`levels`	Specific indices to use as hierarchical index level or levels if keys passed.
`names`	Names for created hierarchical levels if `keys` and/or `levels` passed.
`verify_integrity`	Check new axis in concatenated objects for duplicates and raise exception if so. `False` by default.
`ignore_index`	Do not preserve indices along concatenation axis.

Pandas: Data wrangling
Reshaping and pivoting

`007-Pandas-Data-wrangling.ipynb`

- A different kind of transformations is its reshaping, sometimes called pivoting.

- It consists in transposing rows and columns, modifying data dimensionality.

- The basic reshaping and pivoting operations provided by Pandas are:
  - `stack()`: pivots from the columns in the data to the rows.
  - `unstack()`: pivots from the rows into the columns.
  - `pivot()`: reshapes rows and columns, allowing to transform tables in "long" format to "wide" format in a single step.

Pandas: Data wrangling
Data transformations

`007-Pandas-Data-wrangling.ipynb`

- We have focused on structural modifications. There are other types of transformations which focus on the data:
  - Removing duplicates: `duplicated()`, `drop_duplicates()`.
  - Functional application: `apply()`, `map()`, `applymap()`.
  - Value substitution: `replace()`.
  - Index renaming: `rename()`.
  - Discretization and binning: `cut()`, `qcut()`.
  - Detecting and filtering outliers.
  - Permutation and random sampling: `permutation()`, `take()`.
  - Computing indicators/dummy variables: `get_dummies()`.

Pandas: Data wrangling
String manipulation

`007-Pandas-Data-wrangling.ipynb`

- One of Python’s most popular characteristics is its string manipulation routines.

- The `str` class provides methods for conveniently performing many string operations, such as searching, substitions, splitting, etc.

- Many of these operations accept regular expressions as parameters.

- Pandas adds string functionality, as it allows to apply string operations over data tables automatically handling unknown values.

Pandas: Data wrangling

`str` methods

Method	Description
`count`	Returns the number of non-overlapping occurrences of substring in the string.
`endswidth / startswidth`	Returns `True` if a string ends with suffix / starts with prefix.
`join`	Use string as a delimiter for concatenating a sequence of other strings.
`index`	Return position of first character in substring if found in the string. Raises `ValueError` if not found.

Pandas: Data wrangling

`str` methods

Method	Description
`find`	Like `index()`, but returns `-1` if not found.
`rfind`	Like `find()`, but returns position of last occurrence.
`replace`	Replace occurrences of string with another string.
`strip / rstrip / lstrip`	Trim whitespace, including newlines.
`split`	Break string into list of substrings using passed delimiter.

Pandas: Data wrangling

`str` methods

Method	Description
`lower / upper`	Convert alphabet characters to lowercase or uppercase, respectively.
`ljust / rjust`	Left / right justify. Pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

Pandas: Data wrangling
Regular expressions

`007-Pandas-Data-wrangling.ipynb`

- Regular expressions provide a flexible way to search or match string patterns in text.

- A regular expression is a string containing certain elements, called “special characters”, with a specific semantic.

- A regular expression (*regex*) describes a pattern to match in the text, or a way to manipulate it.

- The `re` module includes functions for regular expression application. It has three types of functions: pattern matching, substitutions, and splitting.

Pandas: Data wrangling
`regex` methods

Method	Description
`findall / finditer`	Return all non-overlapping matching patterns in a string as a list / iterator.
`match`	Match pattern at start of string and optionally segment pattern components into groups. Returns a `match` object, or `None`.
`search`	Scan string for match to pattern; returning a match object if so. The match can be anywhere in the string, as opposed to `match()`.
`split`	Break string into pieces at each occurrence of pattern.
`sub, subn`	Replace all / first n occurrences of pattern with replacement expression.

Pandas: Data wrangling

Vectorized string functions in Pandas

Method	Description
`cat`	Concatenate strings element-wise with optional delimiter.
`contains`	Return boolean array if each string contains pattern.
`count`	Count occurrences of pattern.
`endswith / startswith`	Applies homonym functions in `str` element-wise.
`findall`	Compute list of all occurrences of pattern for each string.

Pandas: Data wrangling
Vectorized string functions in Pandas

Method	Description
`get`	Index into each element (retrieve i-th element).
`join`	Join strings in each element of the Series with passed separator.
`len`	Compute length of each string.
`lower / upper`	Convert cases.
`match`	Use `re.match` with passed regex on each element.
`pad`	Add whitespace to left, right, or both sides of string.

Pandas: Data wrangling
Vectorized string functions in Pandas

Method	Description
`center`	Equivalent to `pad(side="both" ).`
`repeat`	Duplicate values.
`replace`	Replace occurrences of pattern with some other regex.
`slice`	Slice each string in the Series.
`split`	Split strings on delimiter.

`strip /`
`rstrip /`
`lstrip`
	Trim whitespace, including newlines, element-wise.

Pandas: Plotting and visualization

008-Pandas-Plotting.ipynb

- Matplotlib provides a powerful framework for plotting and visualization, but it is a low-level tool.
- Building a plot involves configuring several Python objects without implicit semantics.
- Pandas objects are a centralized storage of data, with semantics at least partially known.
- Pandas objects provide a `plot()` method which builds complex plots in a convenient manner.

Pandas: Plotting and visualization

plot() parameters

Parameter	Description
ax	Matplotlib subplot object to plot on. If nothing passed, uses active subplot.
kind	[‘line’|‘bar’|‘barh’|‘hist’|‘box’|‘kde’|‘density’|‘area’|‘pie’|‘scatter’|‘hexbin’]. Type of chart.
logx / logy	Use logarithmic scale on the X / Y axis.
use_index	Use the object index for tick labels.
rot	Rotation angle of tick labels.
xticks / yticks	Values to use for X / Y axis ticks.

Pandas: Plotting and visualization

`plot()` parameters

Parameter	Description
`xlim / ylim`	X / Y axis limits (specified as `[min,max]`).
`grid`	`[True|False]`. Display axis grid.
`subplots`	`[True|False]`. Plot each `DataFrame` column in a separate subplot.
`sharex / sharey`	`[True|False]`. If `subplots=True`, share the same X / Y axis, linking ticks and limits.
`layout`	Tuple indicating the geometry of subplots to use.
`figsize`	Figure size.

Pandas: Plotting and visualization

`plot()` parameters

Parameter	Description
`title`	Plot title.
`legend`	`[True|False|‘reverse’].` Add a subplot legend.
`style`	Dictionary matching each column with the style to use for plotting it.
`loglog`	Use log scale for both axes.
`fontsize`	Font size to use for ticks.
`colormap`	Color map to index.

Pandas: Plotting and visualization

`plot()` parameters

Parameter	Description
`colorbar`	`[True|False]`. Whether to draw the value legend for `‘scatter’` and `‘hexbin’` plot types).
`table`	If a `Series` or `DataFrame` object is provided, includes it in the plot. Useful to combine plots and tables.
`stacked`	Create stacked plot.
`sort_columns`	`[True|False]`. Whether to sort columns lexicographically before plotting.
`secondary_y`	List of columns that should be referenced to a secondary Y axis.

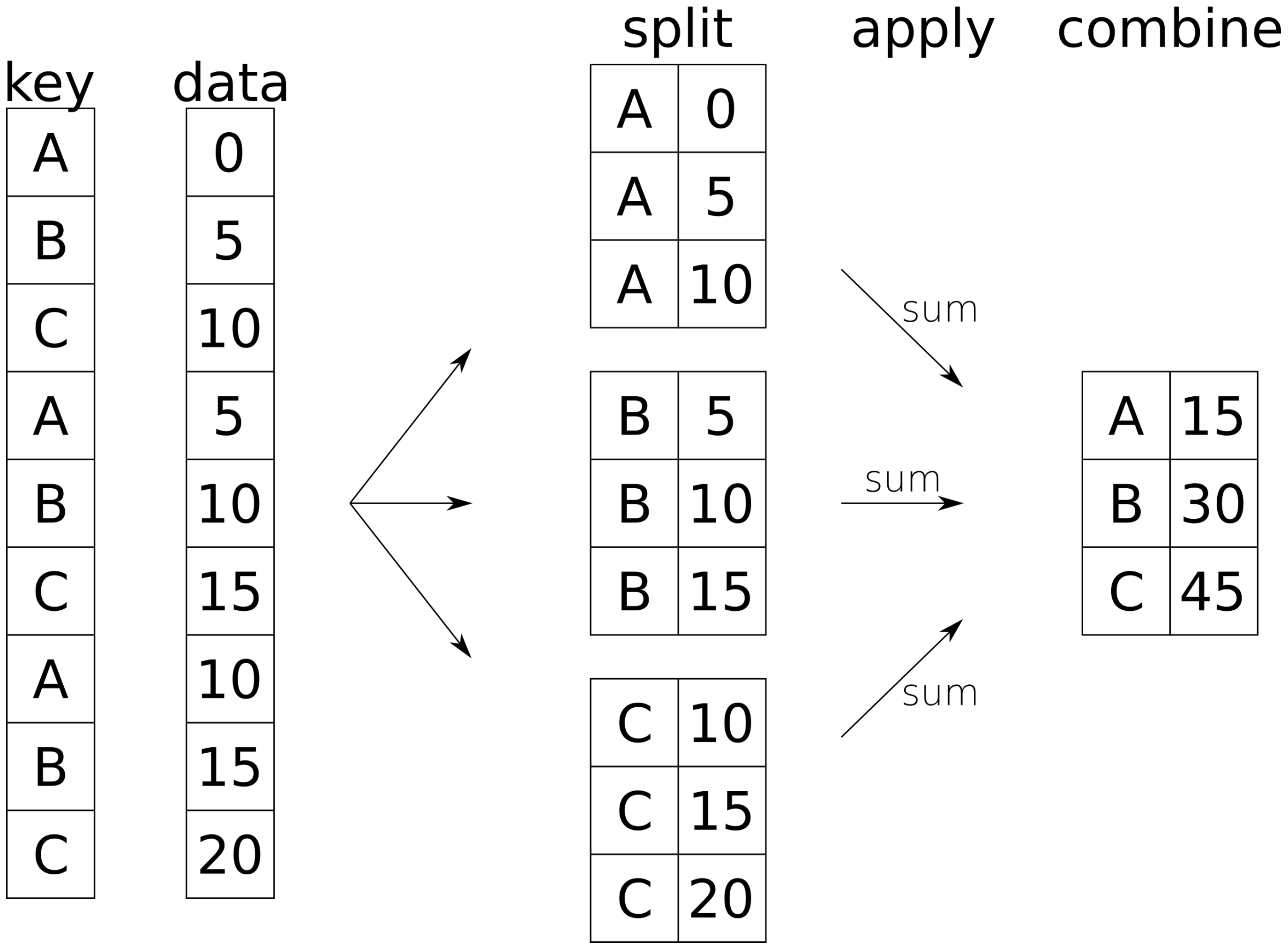
Pandas: Plotting and visualization
`plot()` parameters

Parameter	Description
`mark_right`	`[True|False]`. When a secondary axis is used, automatically add the suffix “`(right)`” to the legends of the series referenced to it.
`**kwds`	Parameters not processed by Pandas will be passed to Matplotlib.

Pandas: Data aggregation

- One of the reasons for the popularity of relational databases and SQL is the ease with which data can be joined, filtered, transformed, and aggregated.
- However, query languages like SQL have limited expressiveness. Pandas allows to implement *split-apply-combine* operations conveniently:
  - Split a Pandas object into pieces using one or more keys.
  - Compute group summary statistics.
  - Apply a varying set of functions to each column of a `DataFrame`.
  - Compute pivot tables and cross-tabulations.
  - Perform quantile analysis and other data-derived group analyses.

Pandas: Data aggregation



key	data
A	0
B	5
C	10
A	5
B	10
C	15
A	10
B	15
C	20

split

A	0
A	5
A	10

B	5
B	10
B	15

C	10
C	15
C	20

apply

sum

sum

sum

combine

A	15
B	30
C	45

Pandas: Data aggregation

`009-Pandas-Aggregation.ipynb`

- A critical aspect of these transformations is how to categorize data.
- A Pandas object is *split* into groups based on one or more keys, applied on a particular axis.
- The Pandas mechanism for this operation is `groupby()`.
- A grouping key can take many forms, e.g.:
  - A list or array of values that is the same length as the axis being grouped.
  - A value indicating a column name in a `DataFrame`.
  - A `dict` or `Series` giving correspondences between the values on the axis being grouped and the group names.
  - A function to be invoked on the axis index or the individual labels of the index.

Pandas: Data aggregation
Aggregation functions

`009-Pandas-Aggregation.ipynb`

- An *aggregation functions* is any transformation which produces a scalar value from an array (also called *reduction*).
- Aggregation functions as implemented by the `GroupBy` class have been optimized and are computed on the original data of the `DataFrame` or `Series`.
- Applicable aggregation functions are not limited to this subset: any function, including user-defined functions, can be applied to a grouped dataset.

Pandas: Data aggregation
`GroupBy` methods

Method	Description
`count`	Number of non-NA values in the group.
`sum`	Sum of non-NA values.
`mean`	Mean of non-NA values.
`median`	Arithmetic median of non-NA values.

Pandas: Agregación
Métodos en `GroupBy`

Método	Descripción
`std / var`	Unbiased standard deviation / variance.
`min / max`	Minimum / maximum of non-NA values.
`prod`	Product of non-NA values.
`first / last`	First / last non-NA value.

Pandas: Group-wise operations

`009-Pandas-Aggregation.ipynb`

- Aggregation is only one kind of group operation: accepts functions that reduce a one-dimensional array to a scalar value.
- In the general case, we want to apply any kind of operation to grouped data.
- This is done using `transform()` and `apply()`:
  - `transform()` broadcasts the result of an aggregation over the original members of the group.
  - `apply()` applies a function to each group and combines the results using `pandas.concat()`.

`009-Pandas-Aggregation.ipynb`

Pandas: Pivot tables and cross-tabulation

- A pivot table is a data summarization tool which aggregates a table by one or more keys, arranging the data in a rectangle with some groups along rows and some along columns.
- It can be built using `groupby()`, but `pivot_table()` provides a more convenient high-level interface.
- A cross-tabulation is a special case of a pivot table that computes group frequencies.
- Could also be built manually using several functions, but `crosstab()` simplifies the process.

Pandas: Time series

```
010-Pandas-Time-series.ipynb
```

- Any dataset which includes observations at many points in time forms a time series.
- Many time series are *fixed frequency*: data points occur at regular intervals.
- Others are *irregular*: without a fixed offset between data points.
- How time series are referred depends on the application. Among others:
    - *Timestamps*: specific instants in time.
    - Fixed *periods*, such as the month of January 2007 or the full year 2010.
    - *Intervals* of time, indicated by start an end timestamps.
    - Elapsed time relative to a particular fixed start time.
- Pandas provides a standard set of time series tools and data algorithms to slice and dice, aggregate, resample, etc.

Pandas: Time series
Base frequencies

Alias	***Offset* type**	**Description**
D	Day	Calendar daily.
B	BusinessDay	Business daily.
H	Hour	Hourly.
T / min	Minute	Minutely.
S	Second	Secondly.
L / ms	Milli	Millisecond.
U	Micro	Microsecond.

Pandas: Time series
Base frequencies

Alias	*Offset* type	Description
`M`	`MonthEnd`	Last calendar day of month.
`BM`	`BusinessMonthEnd`	Last business day of month.
`MS`	`MonthBegin`	First calendar day of month.
`BMS`	`BusinessMonthBegin`	First business day of month.
`W-MON, W-TUE, ...`	`Week`	Weekly on given day of week: `MON`, `TUE`, `WED`, `THU`, `FRI`, `SAT`, or `SUN`.
`WOM-1MON,WOM-1TUE, ...`	`WeekOfMonth`	Generate weekly dates on the first, second, third, or fourth week of the month. For example, `WOM-3FRI` for the 3rd Friday of each month.

Pandas: Time series
Base frequencies

Alias	***Offset* type**	**Description**
`Q-JAN, ...`	`QuarterEnd`	Quarterly dates anchored on last calendar day of each month, for year ending in indicated month.
`BQ-JAN, ...`	`BusinessQuarterEnd`	Quarterly dates anchored on last business day of each month, for year ending in indicated month.
`QS-JAN, ...`	`QuarterBegin`	Quarterly dates anchored on first calendar day of each month, for year ending in indicated month.
`BQS-JAN, ...`	`BusinessQuarterBegin`	Quarterly dates anchored on first business day of each month, for year ending in indicated month.

Pandas: Time series
Base frequencies

Alias	*Offset* type	Description
`A-JAN, ...`	`YearEnd`	Annual dates anchored on last calendar day of given month.
`BA-JAN, ...`	`BusinessYearEnd`	Annual dates anchored on last business day of given month.
`AS-JAN, ...`	`YearBegin`	Annual dates anchored on first calendar day of given month.
`BAS-JAN, ...`	`BusinessYearBegin`	Annual dates anchored on first business day of given month.

Pandas: Time series `resample()` parameters

Parameters	Description
`freq`	String or `DateOffset` indicated desired resample frequency.
`how`	Function name or array function producing aggregated value.
`axis`	Axis to resample on, default to `0` (rows).
`fill_method`	How to interpolate when upsampling (`‘ffill’` or `‘bfill’`).
`closed`	In downsampling, which end of each interval is closed (inclusive). Defaults to `‘right’`.
`label`	In downsampling, how to label the aggregated result, with the right or left bin edge. Defaults to `‘right’`.

Pandas: Time series
`resample()` parameters

Parameters	Description
`loffset`	Time adjustment to the bin labels, such as `‘-1s’` / `Second(-1)` to shift the aggregate labels one second earlier.
`limit`	When forward or backward filling, the maximum number of periods to fill.
`kind`	Aggregate to periods (`‘period’`) or timestamps (`‘timestamp’`); defaults to kind of index the time series has.
`convention`	When resampling periods, the convention (`‘start’` or `‘end’`) for converting the low frequency period to high frequency. Defaults to `‘end’`.

Pandas: Time series
Moving window functions

Función	Descripción
`rolling_count`	Returns number of non-NA observations in each trailing window.
`rolling_sum`	Moving window sum.
`rolling_mean`	Moving window mean.
`rolling_median`	Moving window meadian.
`rolling_std / rolling_var`	Moving window variance / standard deviation. Uses (n-1) denominator.
`rolling_skew / rolling_kurt`	Moving window skewness (3rd moment) / kurtosis (4th moment).

Pandas: Time series
Moving window functions

Function	Description
`rolling_min / rolling_max`	Moving window minimum / maximum.
`rolling_quantile`	Moving window score at percentile / sample quantile.
`rolling_corr / rolling_cov`	Moving window correlation / covariance.
`rolling_apply`	Apply generic array function over a moving window.

Pandas: Time series
Moving window functions

Function	**Description**
`ewma`	Exponentially-weighted moving average.
`ewmstd / ewmvar`	Exponentially-weighted moving standard deviation / variance.
`ewmcorr / ewmcov`	Exponentially-weighted moving correlation / covariance.

Bibliography

- *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Python*. Wes McKinney. O'Reilly Media, 2017 (2ª edición).
- Python Data Science Handbook: Essential Tools for Working with Data. Jake VanderPlas. O'Reilly Media, 2016 (1ª edición).

High level tools

Contents

- Visualization: Seaborn.
- Machine learning: Scikit-Learn.
- Out-of-core computation: Dask.

`10-Seaborn.ipynb`

Seaborn

- Seaborn is a graphics library that improves the design and appearance of Matplotlib plots and includes additional statistical methods:
  - It allows to use themes to unify plot aesthetics.
  - Adds functions to visualize and compare distributions on one and two variables.
  - Linear regression tools.
  - Functions for visualizing data matrices and clustering.
  - Plotting statistical time series.

- Similar to ggplot in R.

11-Scikit-Learn.ipynb

Scikit-Learn

- Extension to SciPy (**Sci**Py Tool**kit**) focused on machine learning.
- Compatible with tabular data from other libraries:
  - NumPy
  - Pandas
  - SciPy.sparse (matrices dispersas).
- Includes algorithms for supervised / unsupervised learning: GLMs, SVMs, kNN, Bayes, decision trees, clustering, etc.
- Cross-validation.
- Grid search of optimal models.
- Parallelization.

12-Dask.ipynb

Dask

- Limited but simple alternative to Spark.
- Restrictions to Big Data in Python:
  - Parallelization: restricted by the GIL.
  - Physical memory.
- Dask provides two main tools:
  - Dynamic task planifier (cluster management).
  - “Big data” collections: trivially perform out-of-core computing.
- Provides limited out-of-core support to other APIs:
  - Pandas
  - Scikit-Learn
- Allows to parallelize computations on a cluster.

Python: Other tools

- Statistical modeling:
  - Statsmodels.
- Big Data:
  - Hadoop.
  - Spark.
- Deep Learning:
  - TensorFlow.
  - PyTorch.
- GPU data science:
  - RAPIDS.

Bibliography

- Python Data Science Handbook: Essential Tools for Working with Data. Jake VanderPlas. O’Reilly Media, 2016 (1ª edición).
- Data Science with Python and Dask. Jesse C. Daniel. Manning Publications, 2019 (1ª edición).