Introduction to Python

Karolos POTAMIANOS

University of Oxford (UK)

February 23, 2021

European School in Instrumentation for Particle and Astroparticle Physics (ESIPAP)

European Scientific Institute, Archamps, France

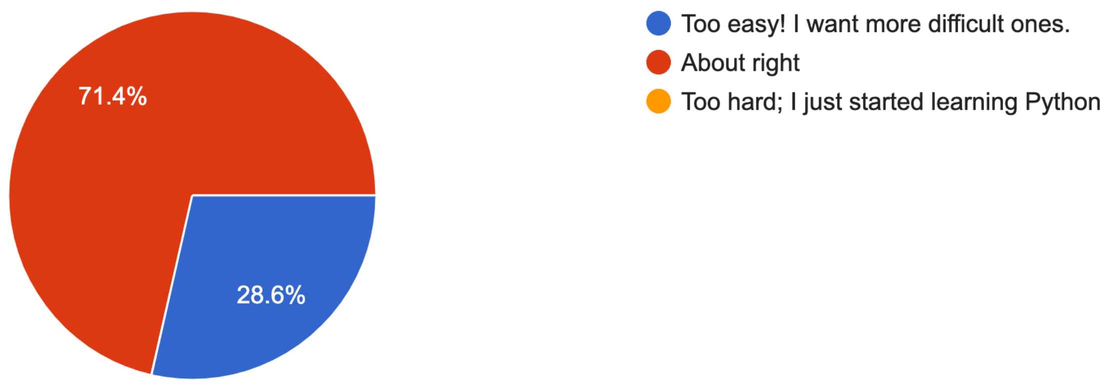
(ONLINE)

Your feedback

- Thanks!

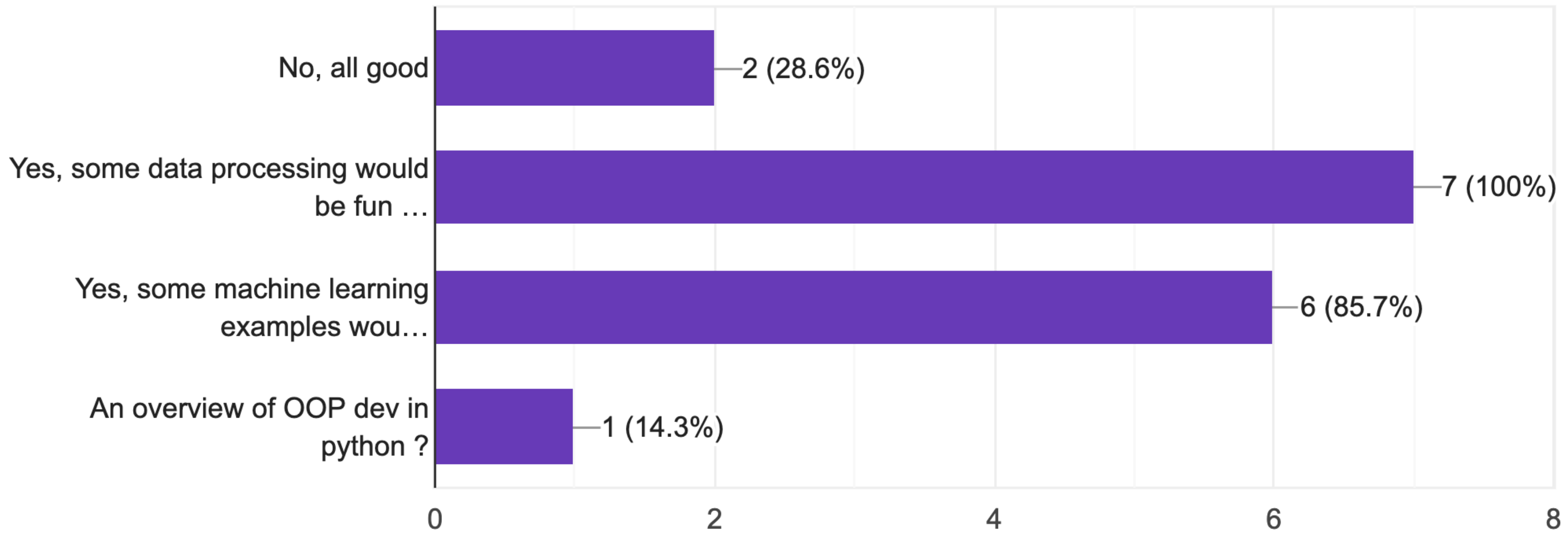
How is the level of the exercises ?

7 responses



Are there some aspects you'd like me to go over (besides object-oriented programming, which we'll do tomorrow) ? [Feel free to add your request in the Other field]

7 responses



Objects in Python

Object Oriented Programming (OOP)

- OOP refers to a type of computer programming in which programmers define the **data type of a data structure** and the **types of operations** (**methods**) that can be applied to the data structure
- Classes provide a means of **bundling data and functionality together**
- Creating a new class creates a **new type of object**, allowing **new instances** of that type to be made
- Each instance can have **attributes** attached to it for **maintaining its state**
- Instances can also have methods (defined by its class) to **modify its state**
- The **class inheritance** mechanism allows multiple **base** classes, a **derived** class can **override any methods of its base class** or classes, and a **method can call the method of a base class with the same name**.

Some definitions

- **Class**: A user-defined prototype for an **object** that defines a set of attributes that characterize any object of the class. The attributes are data members (class variables and instance variables) and methods, accessed via **dot notation**.
- **Class variable**: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables are.
- **Instance variable**: A variable that is defined inside a method and belongs only to the current instance of a class.
- **Method**: A special kind of function that is defined in a class definition.
- **Instance**: An individual object of a certain class. An object obj that belongs to a class Circle, for example, is an instance of the class Circle.
- **Instantiation**: The creation of an instance of a class.

Class and Instance Variables

Python Classes

```
class Dog:
    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name     # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                   # shared by all dogs
'canine'
>>> e.kind                   # shared by all dogs
'canine'
>>> d.name                   # unique to d
'Fido'
>>> e.name                   # unique to e
'Buddy'
```

Class Inheritance

Class Inheritance

```
Class Base1(object): # <class 'object'> is the root of all classes
    <statement-1>
    .
    <statement-N>

class DerivedClassName(Base1):
    <statement-1>
    .
    <statement-N>

Class Base2(object):
    <statement-1>
    .
    <statement-N>

class DerivedClassName(Base1, Base2):  # Multiple inheritance
    <statement-1>
    .
    <statement-N>
```

Class Inheritance

Class Inheritance

```
class MyClass1(object):
    def __init__(self, foo):
        self.foo = foo
    def print(self):
        print(self.foo)
    def hello(self):
        print('hello %s' % self.foo)

class MyClass2(MyClass1):
    def __init__(self, foo, bar):
        super().__init__(foo)     # call the parent constructor
        self.bar = bar
    def print(self):              # the method is overridden
        super().print()           # call the parent method
        print(self.bar)
```

Class Inheritance

Class Inheritance

```
>>> # Previous slide saved as MyModule.py in current folder
>>> import MyModule
>>> x = MyModule.MyClass1("Hello")
>>> y = MyModule.MyClass2("One", "Two")
>>> x.print()
Hello
>>> y.print()
One
Two
```

Class Properties

Class Properties

```
class Celsius:
    def __init__(self, temperature = 0):
        self.temperature = temperature

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    def get_temperature(self):
        print("Getting value")
        return self._temperature

    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value

    temperature = property(get_temperature,set_temperature)
```

Class Properties

Class Properties

```
>>> from Celsius import Celsius # To avoid typing Celsius.Celsius
>>> x = Celsius()
Setting value
>>> x.set_temperature(10)
Setting value
>>> x.get_temperature()
Getting value
10
>>> x.to_fahrenheit()
Getting value
50.0
>>> x.set_temperature(-500)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/karolos/CERNbox/ESIPAP/Python/Celsius.py", line 14, in
set_temperature
    raise ValueError("Temperature below -273 is not possible")
ValueError: Temperature below -273 is not possible
```

Operator Overloading

complex.py

```
class complex:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __add__(self, other):
        return self.a + other.a, self.b + other.b

    def __str__(self):
        return self.a, self.b
```

Operator Overloading

```
>>> from complex import complex
>>> Ob1 = complex(1, 2)
>>> Ob2 = complex(2, 3)
>>> Ob3 = Ob1 + Ob2
>>> print(Ob3)
(3, 5)
>>>
```

Another exercise

Exercise

Write a vector class (arbitrary dimension)

- **Create the module Vector**
- **Create the class Vector**
- **Write the constructor [`def __init__(self, dim)`]**
- **Overload operators**

Use the internet for help, but don't google vector class for an answer

Useful Python Libraries

import os

Miscellaneous operating system interfaces

- More info: https://docs.python.org/3/library/os.html

Lots of functions of the POSIX standard:

- mkdir, rmdir, remove, chmod, etc.
- environ[], setenv(), getenv()
- system(), popen() … to execute shell commands

- `import os.path` for path manipulations (exists, is_dir, etc.)
  - See https://docs.python.org/3/library/os.path.html

Other (System) Libraries

- `import glob`: file wildcards
- `import re`: regular expressions
- `import math`: mathematical functions
- `import random`: random number generation
- `import urllib`: fetching resources from the internet
- `import time, datetime`: time manipulation
- `import zlib`: compression

Libraries Provided by 3rd Parties

- There is a very broad ecosystem of Python libraries provided by third parties. Here we just name a few.

- SciPy: Python-based ecosystem of open-source software for mathematics, science, and engineering. See https://www.scipy.org
    - NumPy: base for N-dimensional array package
    - SciPy: fundamental library for scientific computing
    - Matplotlib: for 2D/3D plotting
    - IPyton: enhanced interactive console
    - Sympy: symbolic mathematics
    - Pandas: Data structure and analysis

Numpy :: pip install numpy

- Numpy: the fundamental package for scientific computing with Python
- Powerful N-dimensional arrays: fast and versatile, the NumPy vectorization, indexing, and broadcasting concepts are de-facto standards
- Numerical computing tools: comprehensive mathematical functions, random number generators, linear algebra routines, Fourier transforms, and more
- Interoperable: supports a wide range of hardware and computing platforms, and plays well with distributed, GPU, and sparse array libraries
- Performant: well-optimized C code. Enjoy the flexibility of Python with the speed of compiled code.
- Easy to use: high level syntax makes it accessible and productive for programmers from any background or experience level
- Open source: BSD license

Numpy arrays

Numpy

```
>>> import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class 'numpy.ndarray'>
```

Numpy arrays : a note on syntax

Numpy

```
>>> a = np.array(1,2,3,4)     # WRONG
Traceback (most recent call last):
   ...
TypeError: array() takes from 1 to 2 positional arguments but 4 were
given
>>> a = np.array([1,2,3,4])   # RIGHT
```

Numpy

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Numpy array initialisation

Numpy

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be
specified
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                                 # uninitialized
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],  # may vary
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

Numpy

Numpy

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
>>> np.arange( 0, 2, 0.3 )                 # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
>>> from numpy import pi
>>> np.linspace( 0, 2, 9 )                 # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
>>> x = np.linspace( 0, 2*pi, 100 )        # useful to evaluate function at lots
of points
>>> f = np.sin(x)
```

Numpy

Numpy

```
>>> a = np.arange(6)                          # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)            # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)          # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Numpy : smart printing

Numpy

```
>>> print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100,100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

No time to cover everything … check https://numpy.org

Pandas :: pip install pandas

- A fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language
- Used very often, and (IMO) a must-know
- Locally you install them using pip

Importing Pandas

```
 (venv) $ pip install pandas numpy
$ python3
...
>>> # Customary imports for Numpy and Pandas
>>> import numpy as np
>>> import pandas as pd
```

- Fundamental data structures of Pandas are **Series** and **DataFrame**

Pandas Series

Series

```
>>> s = pd.Series(np.random.randn(4), index=["a", "b", "c", "d"])
>>> s
a     0.959893
b     0.176460
c    -0.433848
d    -0.469920
dtype: float64
>>> s.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> pd.Series(np.random.randn(5))
0     0.669462
1    -0.599999
2     0.956523
3    -0.300907
4     2.535160
dtype: float64
```

Series is a 1D labelled array capable of holding any data type (integers, strings, floating point numbers, objects, etc.). The axis labels are collectively referred to as the index.

Pandas Series from a dictionary or a scalar value

Series

```
>>> d = {"b": 1, "a": 0, "c": 2}
>>> pd.Series(d)
b     1
a     0
c     2
dtype: int64
>>> pd.Series(d, index=["b", "c", "d", "a"])
b     1.0
c     2.0
d     NaN
a     0.0
dtype: float64
>>> pd.Series(5.0, index=["a", "b", "c", "d", "e"])
a     5.0
b     5.0
c     5.0
d     5.0
e     5.0
dtype: float64
```

Using Pandas Series

Series

```
>>> s[0]
0.9598926249543338
>>> s[:3]
a    0.959893
b    0.176460
c   -0.433848
dtype: float64
>>> s[s > s.median()]
a    0.959893
b    0.176460
dtype: float64
>>> s[[3,2]]
d   -0.469920
c   -0.433848
dtype: float64
>>> np.exp(s)
a    2.611416
b    1.192987
c    0.648011
d    0.625052
dtype: float64
```

Using Pandas Series

Series

```
>>> s.dtype
dtype('float64')
>>> s.array
<PandasArray>
[ 0.9598926249543338, 0.17646021129740513, -0.4338476013125361,
 -0.4699200545656808]
Length: 4, dtype: float64
>>> s.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> s.to_numpy()
array([ 0.95989262,  0.17646021, -0.4338476 , -0.46992005])
```

Using Pandas Series

Series

```
>>> s["a"]
0.9598926249543338
>>> s["d"] = 12.0
>>> s
a     0.959893
b     0.176460
c    -0.433848
d    12.000000
dtype: float64
>>> "d" in s
True
>>> "e" in s
False
>>> s["f"]
Traceback (most recent call last):
...
KeyError: 'f'
>>> s.get("f")
>>> s.get("f", np.nan)
nan
```

Using Pandas Series

Series

```
>>> s+s
a     1.919785
b     0.352920
c    -0.867695
d    24.000000
dtype: float64
>>> s*2
a     1.919785
b     0.352920
c    -0.867695
d    24.000000
dtype: float64
>>> s[1:] + s[:-1] # Operations done correctly based on index label
a         NaN
b    0.352920
c   -0.867695
d         NaN
dtype: float64
```

Pandas DataFrame

DataFrame

```
>>> d = {
...     "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
...     "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c",
"d"]),
... }
>>> df = pd.DataFrame(d)
>>> df
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0
```

DataFrame is a 2D labelled data structure with columns of potentially different types.

Pandas DataFrame from list

DataFrame

```
>>> d = {
...     "one": pd.Series([1.0, 2.0, 3.0], index=["a", "b", "c"]),
...     "two": pd.Series([1.0, 2.0, 3.0, 4.0], index=["a", "b", "c",
"d"]),
... }
>>> pd.DataFrame(d, index=["d", "b", "a"])
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0
>>> pd.DataFrame(d, index=["d", "b", "a"], columns=["two", "three"])
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
>>> df.index
Index(['a', 'b', 'c', 'd'], dtype='object')
>>> df.columns
Index(['one', 'two'], dtype='object')
```

Pandas DataFrame

DataFrame

```
>>> d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0,
>>> pd.DataFrame(d)
   one  two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0
>>> pd.DataFrame(d, index=["a", "b", "c", "d"])
   one  two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

Pandas DataFrame from a structured array

DataFrame

```
>>> np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])
array([(0, 0., b''), (0, 0., b'')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
>>> data = np.zeros((2,), dtype=[("A", "i4"), ("B", "f4"), ("C", "a10")])
>>> data[:] = [(1, 2.0, "Hello"), (2, 3.0, "World")]
>>> pd.DataFrame(data)
   A    B         C
0  1  2.0  b'Hello'
1  2  3.0  b'World'
>>> pd.DataFrame(data, index=["first", "second"])
        A    B         C
first   1  2.0  b'Hello'
second  2  3.0  b'World'
>>> pd.DataFrame(data, columns=["C", "A", "B"])
          C  A    B
0  b'Hello'  1  2.0
1  b'World'  2  3.0
```

Pandas DataFrame from list of dicts

DataFrame

```
>>> data2 = [{"a": 1, "b": 2}, {"a": 5, "b": 10, "c": 20}]
>>> pd.DataFrame(data2)
   a   b     c
0  1   2   NaN
1  5  10  20.0
>>> pd.DataFrame(data2, index=["first", "second"])
        a   b     c
first   1   2   NaN
second  5  10  20.0
>>> pd.DataFrame(data2, columns=["a", "b"])
   a   b
0  1   2
1  5  10
```

Pandas DataFrame from dict of tuples

DataFrame

```
>>> pd.DataFrame(
...     {
...         ("a", "b"): {("A", "B"): 1, ("A", "C"): 2},
...         ("a", "a"): {("A", "C"): 3, ("A", "B"): 4},
...         ("a", "c"): {("A", "B"): 5, ("A", "C"): 6},
...     }
... )
       a
       b  a  c
A B    1  4  5
  C    2  3  6
```

Creates a **multiindexed** frame

Operations on DataFrame

DataFrame

```
>>> df["one"]
a     1.0
b     2.0
c     3.0
d     NaN
Name: one, dtype: float64
>>> df["three"] = df["one"] * df["two"]
>>> df["flag"] = df["one"] > 2
>>> df
   one  two  three   flag
a  1.0  1.0    1.0  False
b  2.0  2.0    4.0  False
c  3.0  3.0    9.0   True
d  NaN  4.0    NaN  False
```

Operations on DataFrame

DataFrame

```
>>> del df["two"]
>>> three = df.pop("three")
>>> df
   one   flag
a  1.0  False
b  2.0  False
c  3.0   True
d  NaN  False
>>> df["foo"] = "bar"
>>> df
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
```

Operations on DataFrame

DataFrame

```
>>> df
   one   flag  foo
a  1.0  False  bar
b  2.0  False  bar
c  3.0   True  bar
d  NaN  False  bar
>>> df["one_trunc"] = df["one"][:2]
>>> df
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
```

Operations on DataFrame

DataFrame

```
>>> df
   one   flag  foo  one_trunc
a  1.0  False  bar        1.0
b  2.0  False  bar        2.0
c  3.0   True  bar        NaN
d  NaN  False  bar        NaN
>>> df.insert(1, "bar", df["one"])
>>> df
   one  bar   flag  foo  one_trunc
a  1.0  1.0  False  bar        1.0
b  2.0  2.0  False  bar        2.0
c  3.0  3.0   True  bar        NaN
d  NaN  NaN  False  bar        NaN
```

Reading in data from a CSV file

- DataFrames are really powerful at reading data from various sources, such as CSV (comma-separated values) files

DataFrame

```
>>> import os
>>> os.system("wget https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data >& /dev/null")
>>> iris = pd.read_csv("iris.data", names=["SepalLength", "SepalWidth", "PetalLength", "PetalWidth", "Name"])
>>> iris.head()
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
2          4.7         3.2          1.3         0.2  Iris-setosa
3          4.6         3.1          1.5         0.2  Iris-setosa
4          5.0         3.6          1.4         0.2  Iris-setosa
```

Operations on DataFrame

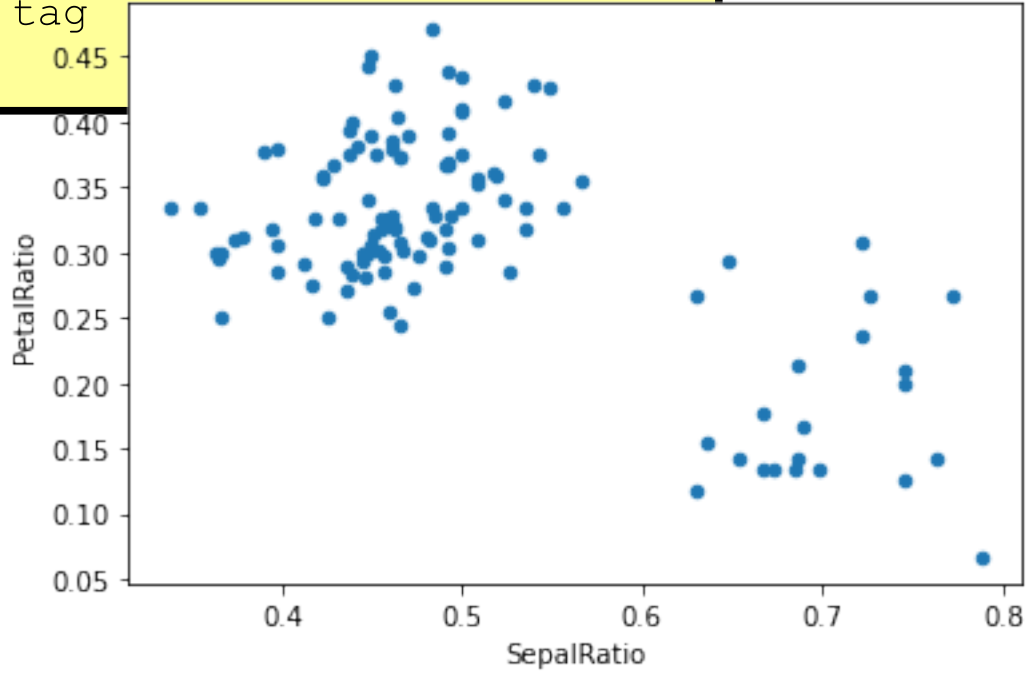
DataFrame

```
>>> iris.columns
Index(['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Name'],
dtype='object')
>>> iris.assign(sepal_ratio=iris["SepalWidth"] / iris["SepalLength"]).head()
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
>>> iris.assign(sepal_ratio=lambda x: (x["SepalWidth"] / x["SepalLength"])).head()
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name  sepal_ratio
0          5.1         3.5          1.4         0.2  Iris-setosa     0.686275
1          4.9         3.0          1.4         0.2  Iris-setosa     0.612245
2          4.7         3.2          1.3         0.2  Iris-setosa     0.680851
3          4.6         3.1          1.5         0.2  Iris-setosa     0.673913
4          5.0         3.6          1.4         0.2  Iris-setosa     0.720000
>>> iris.head(2)
   SepalLength  SepalWidth  PetalLength  PetalWidth         Name
0          5.1         3.5          1.4         0.2  Iris-setosa
1          4.9         3.0          1.4         0.2  Iris-setosa
```

Operations on DataFrame

DataFrame

```
>>> import matplotlib
>>> (
...         iris.query("SepalLength > 5")
...         .assign(
...             SepalRatio=lambda x: x.SepalWidth / x.SepalLength,
...             PetalRatio=lambda x: x.PetalWidth / x.PetalLength,
...        )
...        .plot(kind="scatter", x="SepalRatio", y="PetalRatio")
... )
Unable to revert mtime: /Library/Fonts
Fontconfig warning: ignoring UTF-8: not a valid region tag
<AxesSubplot:xlabel='SepalRatio', ylabel='PetalRatio'>
```



More on this dataset later …

Operations on DataFrame

DataFrame

```
>>> dates = pd.date_range("20130101", periods=6)
>>> dates
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
>>> df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
>>> df
                   A         B         C         D
2013-01-01 -0.616776  2.255949 -0.804261  0.136389
2013-01-02  2.036988  0.133639 -2.194401 -1.314238
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
>>> df.tail(3)
                   A         B         C         D
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
```

Too many things to do with DataFrame …

Operations on DataFrame

DataFrame

```
>>> df
                   A         B         C         D
2013-01-01 -0.616776  2.255949 -0.804261  0.136389
2013-01-02  2.036988  0.133639 -2.194401 -1.314238
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
>>> df.iloc[3]
A   -1.496272
B   -2.286047
C   -0.255013
D    0.302078
Name: 2013-01-04 00:00:00, dtype: float64
>>> df.iloc[3:5, 0:2]
                   A         B
2013-01-04 -1.496272 -2.286047
2013-01-05  0.527517  0.704457
>>> df[df["A"] > 0]
                   A         B         C         D
2013-01-02  2.036988  0.133639 -2.194401 -1.314238
2013-01-05  0.527517  0.704457 -0.248928  0.040991
2013-01-06  1.990637  0.713619  0.499214  1.528811
```

A word on missing data

DataFrame

```
>>> df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ["E"])
>>> df1.loc[dates[0] : dates[1], "E"] = 1
>>> df1
                   A         B         C         D    E
2013-01-01 -0.616776  2.255949 -0.804261  0.136389  1.0
2013-01-02  2.036988  0.133639 -2.194401 -1.314238  1.0
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581  NaN
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078  NaN
>>> df1.dropna(how="any")
                   A         B         C         D    E
2013-01-01 -0.616776  2.255949 -0.804261  0.136389  1.0
2013-01-02  2.036988  0.133639 -2.194401 -1.314238  1.0
>>> df1.fillna(value=5)
                   A         B         C         D    E
2013-01-01 -0.616776  2.255949 -0.804261  0.136389  1.0
2013-01-02  2.036988  0.133639 -2.194401 -1.314238  1.0
2013-01-03 -1.955609  0.814910 -0.450114 -2.416581  5.0
2013-01-04 -1.496272 -2.286047 -0.255013  0.302078  5.0
>>> pd.isna(df1).tail(2)
                A      B      C      D     E
2013-01-03  False  False  False  False  True
2013-01-04  False  False  False  False  True
```

A word on missing data

- As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data.
- While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object.
- In many cases, however, the Python None will arise and we wish to also consider that “missing” or “not available” or “NA”.
- To consider inf and -inf as “NA”, need to set `pandas.options.mode.use_inf_as_na = True`
- Note: Starting from pandas 1.0, an experimental `pd.NA` value (singleton) is available to represent scalar missing values.

Stats in Pandas (excluding missing data)

DataFrame

```
>>> df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))
>>> df
                   A         B         C         D
2013-01-01 -0.874710 -0.479047  0.349441 -0.069385
2013-01-02  0.815037  2.159031 -0.230224  0.001993
2013-01-03  0.393732 -0.418193  0.638911 -0.467236
2013-01-04 -0.744395 -1.551088  0.622943  0.622890
2013-01-05  0.239997 -0.971747 -1.119259 -0.512740
2013-01-06 -0.723792  0.131198  0.992990  0.421122
>>> df.mean()
A   -0.149022
B   -0.188308
C    0.209134
D   -0.000559
dtype: float64
>>> df.mean(1)
2013-01-01   -0.268425
2013-01-02    0.686459
2013-01-03    0.036804
2013-01-04   -0.262413
2013-01-05   -0.590937
2013-01-06    0.205380
Freq: D, dtype: float64
```

Grouping

DataFrame

```
>>> df["E"] = "Good"
>>> df.loc[dates[3]:dates[5], "E"] = "Bad"
>>> df
                   A         B         C         D     E
2013-01-01 -0.874710 -0.479047  0.349441 -0.069385  Good
2013-01-02  0.815037  2.159031 -0.230224  0.001993  Good
2013-01-03  0.393732 -0.418193  0.638911 -0.467236  Good
2013-01-04 -0.744395 -1.551088  0.622943  0.622890   Bad
2013-01-05  0.239997 -0.971747 -1.119259 -0.512740   Bad
2013-01-06 -0.723792  0.131198  0.992990  0.421122   Bad
>>> df.groupby("E").sum()
             A         B         C         D
E
Bad  -1.228191 -2.391636  0.496674  0.531271
Good  0.334058  1.261791  0.758128 -0.534628
>>> df.groupby("E").mean()
             A         B         C         D
E
Bad  -0.409397 -0.797212  0.165558  0.177090
Good  0.111353  0.420597  0.252709 -0.178209
```

One could keep going for a very long time … check doc @ https://pandas.pydata.org

Libraries Provided by 3rd Parties (2)

- Of course, there are also the machine learning libraries…

- **Tensorflow:** TensorFlow is an end-to-end python machine learning library for performing high-end numerical computations: can handle deep neural networks for image recognition, handwritten digit classification, recurrent neural networks, NLP (Natural Languae Processing), word embedding, etc.
- **Keras:** leading open-source Python library written for constructing neural networks and machine learning projects.
- **Scikit-learn:** another prominent open-source Python machine learning library with a broad range of clustering, regression and classification algorithms.
- **PyTorch:** deep neural networks and Tensor computation with GPU acceleration are the two high-end features of the PyTorch
- **Theano:** aims to boost development time and execution time of ML apps, particularity in deep learning algorithms. (Syntax is not beginner-friendly.)

A few examples with Scikit Learn and Keras/TF

- In the tutorial, we'll cover some basic examples from machine learning (ML) and deep learning (DL) though these are beyond the scope of this course on Python.
- Yet, they allow you to understand how Python can be used in a production environment.
- Note that these examples come from an ML/DL course, and I don't have time to cover the basics here, but you can read up online.
- Examples (in GitLab):
  - Iris-ML-Example.ipynb: Using ML to classify plants (Iris)
  - Iris.ipynb : using a neural network to classify plants (Iris)
  - MNIST-TF-Keras-CNN.ipynb : Convolutional Neural Networks with MNIST dataset
  - MNIST-TF-Keras-NN-Basic.ipynb: Neural Networks with MNIST dataset

Quantum Computing Simulation

- Quantum computing offer exciting perspectives for the future (of course, provided that they can deliver), but the hardware is not yet widely available. Until then, we have to rely on **simulation** to try them out.

Two very popular Python frameworks:

- PennyLane (https://pennylane.ai) is a cross-platform Python library for differentiable programming of quantum computers.
  - Essentially allows training a quantum computer the same way as a neural network.
- Tensorflow Quantum (https://www.tensorflow.org/quantum) is a library for hybrid quantum-classical machine learning

- Unfortunately I don’t have time to discuss these (they go well beyond this class) but you may have some fun exploring this

Conclusion

- Python is an interpreted, high-level, general-purpose programming language.
- Python's design philosophy emphasizes code readability with its notable use of significant whitespace.
- Python is meant to be an easily readable language. It is easy to learn.
- Python is slower than other languages but is excellent at interfacing with them to write nice **user code**.
- Python's name is derived from the British comedy group Monty Python, whom Python creator enjoyed while developing the language.

Python

Appeared in	1991; 30 years ago
Designed by	Guido van Rossum
Stable release	3.9.1 (& 2.7.18)
URL	http://www.python.org/
OS	cross-platform

Next Steps

Like a real-life language, one needs to practice to gain experience with Python. Luckily there are plenty of resources online to achieve this.

See for example https://www.practicepython.org

The documentation is a great reference: https://docs.python.org/3/

Enjoy programming in Python!

Thank you