Python Tutorial – Part I: Introduction

Mark A. Austin

University of Maryland

austin@umd.edu
ENCE 688P, Spring Semester 2022

January 18, 2023

Overview

Introduction

What is Python?

What is Python?

The Origins of Python

The Python programming language was initially written by Guido van Rossum in the late 1980s and first released in the early '90s. Its design borrows features from C, C++, Smalltalk, etc.

The name Python comes from Monty Python's Flying Circus.

Version 0.9 was released in February 1991. Fast forward to 2022, and we are up to Version 3.11.

What is Python?

Features:

- Designed for quick-and-dirty scripts, reusable modules, very large systems.
- Object-oriented. Very well-designed. Well documented.
- Large library of standard modules and third-party modules.
- Works on Unix, Mac OS X and Windows.
- Python is both a compiled and interpreted language. Python source code is compiled into a bytecode format.
- Integration with external C and Java code (Jython).

What is Python?

Strengths of Python:

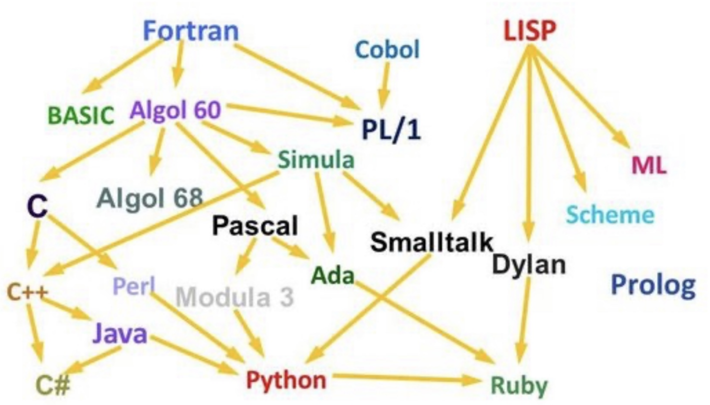
- Open source. Compared to C and Java, it's easy to learn.
- Provides an approximate superset of MATLAB functionality.
- Modern language with good support for object-oriented program development.

Third-Party Modules:

- NumPy is a language extension that defines the numerical array and matrix type and basic operations on them.
- SciPy uses numpy to do advanced math, signal processing, optimization, statistics, etc.
- Matplotlib provides easy-to-use plotting Matlab-style.

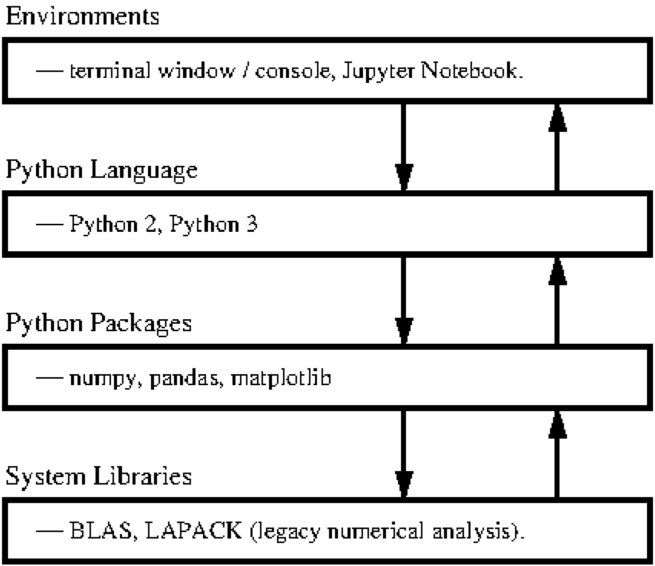
What is Python?

Graph of Feature Dependencies Among Computer Languages



Python Language: Borrows from C++, Java, Smalltalk, ..

Framework for Scientific Computing



Program Development with Python

First Steps: Working with the Terminal

Terminal Window (Console)

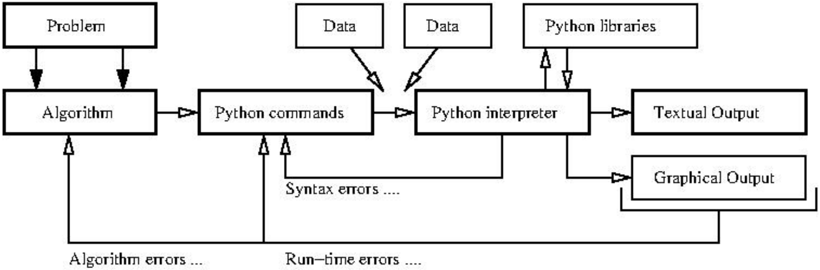
The standard approach runs a program directly through the Python intepreter.

Terminal — Python — 112×26

```
/Users/austin 872>> python3
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 26 2018, 23:26:24)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> a = [ 1, 2, 3, 4, 5, 6]
>>> print(a)
[1, 2, 3, 4, 5, 6]
>>> print(type(a))
<class 'list'>
>>> b = [ (1, 2), (3, 4), (5, 6) ]
>>> print(b)
[(1, 2), (3, 4), (5, 6)]
>>> print( type(b) )
<class 'list'>
>>> import numpy as np
>>> c = np.array(b)
>>> print(c)
[[1 2]
 [3 4]
 [5 6]]
>>> print( type(c) )
<class 'numpy.ndarray'>
>>> 
```

First Steps: Working with the Terminal

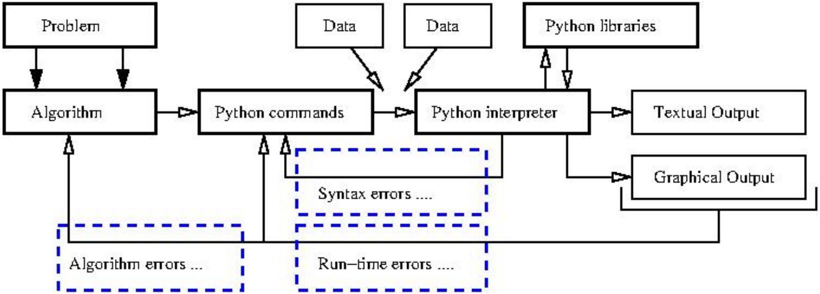
Program Development in the Terminal Window:



Step-by-Step Procedure:

1. Write, compile, fix, run, fix, run, validate → success!

First Steps: Fixing Mistakes



1. Syntax Errors: Check your typing ...
2. Runtime Errors: Program runs, but you have divide by zero and/or NaNs, etc.
3. Algorithm Errors: Does your program solve the right problem?

First Steps: Program Evaluation

Program Evaluation

- Robustness (does it work?)
- Accuracy and Efficiency (speed).
- Ease of Implementation (cost).

Things to Learn:

- How are numbers stored inside the computer?
- How do variables work?
- How do vectors and matrices work?
- How do list, dictionaries and sets work?
- What's in the Python Programming Language?
- How to apply Python to solution of numerical problems?
- Where can I go for help?

Integrated

Development Environments

(Simplifying Program Development)

Integrated Development Environments

Integrated Development Environments

An Integrated Development Environment (IDE) is a software application that provides comprehensive support to computer programmers for software development.

State-of-the-art IDEs provide tools for:

- Syntax highlighting, editing source code, automation of program build, and code debugger.
- Program compilation (interpretation) and execution (run).

Two IDE's for Python:

- Visual Studio Code (for program development).
- Jupyter Notebook (web-based authoring of python documents).

Visual Studio Code

Visual Studio Code (vscode)

Visual Studio Code (vscode) is a source code editor for Windows, Linux and macOS. Features include support for debugging, syntax highlighting, intelligent code completion and code refactoring.

Standard Use Cases:

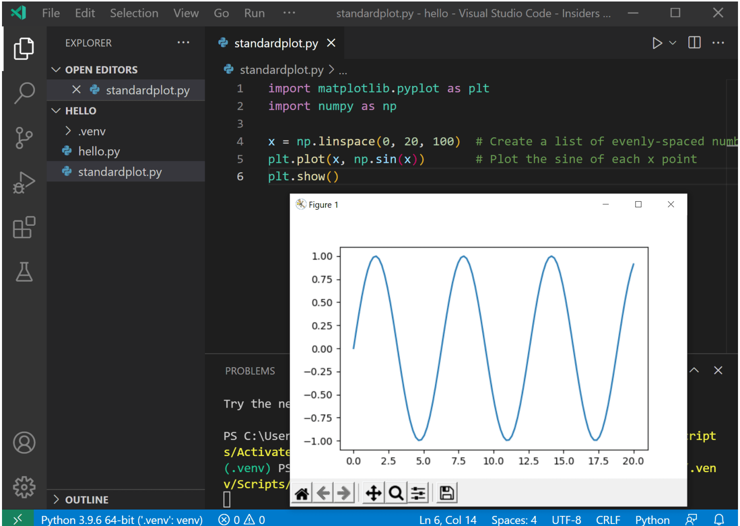
- Edit, debug, run, debug, run, test.
- Develop desktop apps.
- Numerical and scientific computing.

Advanced Use Cases:

- Deploy code to the cloud (Github).

Visual Studio Code

Graphical Interface



Jupyter Notebook

Jupyter Notebook (Web-based Application)

Web-based authoring of documents that combine live code with narrative text, equations and visualization.

To install Jupyter Notebook:

```
prompt >> pip3 install jupyter
```

To run Jupyter Notebook:

```
prompt >> jupyter notebook
```

Jupyter Notebook

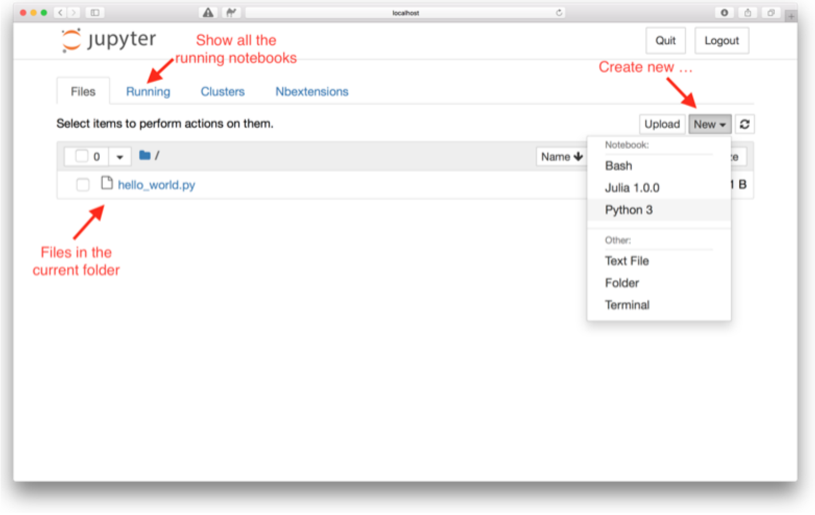
Use Cases:

- Data cleaning and transformation.
- Numerical simulation.
- Statistical modeling.
- Data visualization.
- Machine learning.

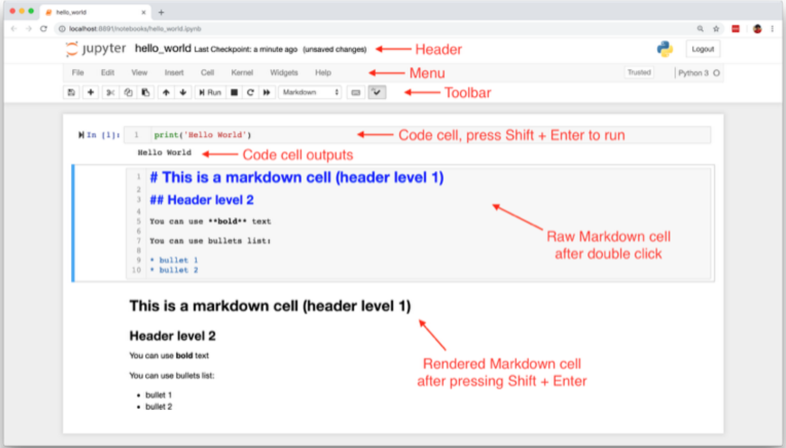
Jupyter Notebook File Format:

- File format is JSON-based with extension .ipynb (named after projects predecessor IPython).
- Supports documents containing text, source code, rich media data and metadata.

Jupyter Notebook User Interface



Jupyter Notebook User Interface



Jupyter Notebook Cells and Code Execution

Jupyter Notebook Cells:

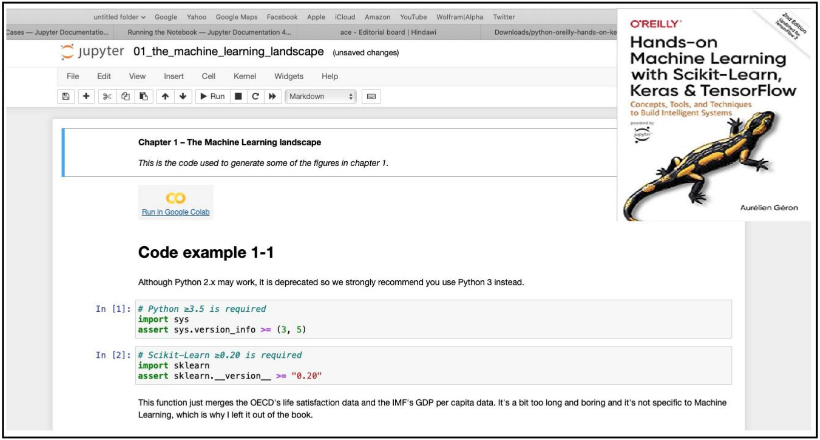
- **Code Cells:** Allows for development and editing of new code, with syntax highlighting and tab completion.
- **Markdown Cells:** Document the computational process with the Markdown language (a simple way to perform text markup). Can also include mathematics with LaTeX notion.
- **Raw Cells:** Provide a place in which you can write output directly.

Code Execution:

- When a code cell is executed, the code is sent to the kernel associated with the code.
- Results are returned to the computation and then displayed.

Jupyter Notebook and Machine Learning

Jupyter Notebook (Machine Learning with TensorFlow)

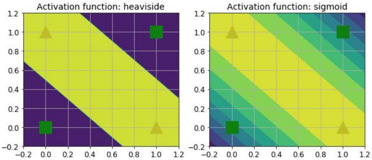


Jupyter Notebook and Machine Learning

Jupyter Notebook (Machine Learning with TensorFlow)

In [7]:
```python
def heaviside(z):
    return (z >= 0).astype(z.dtype)

def mlp_xor(x1, x2, activation=heaviside):
    return activation(-activation(x1 + x2 - 1.5) + activation(x1 + x2 - 0.5) - 0.5)
```

In [8]:
```python
x1s = np.linspace(-0.2, 1.2, 100)
x2s = np.linspace(-0.2, 1.2, 100)
x1, x2 = np.meshgrid(x1s, x2s)

z1 = mlp_xor(x1, x2, activation=heaviside)
z2 = mlp_xor(x1, x2, activation=sigmoid)

plt.figure(figsize=(10,4))

plt.subplot(121)
plt.contourf(x1, x2, z1)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Activation function: heaviside", fontsize=14)
plt.grid(True)

plt.subplot(122)
plt.contourf(x1, x2, z2)
plt.plot([0, 1], [0, 1], "gs", markersize=20)
plt.plot([0, 1], [1, 0], "y^", markersize=20)
plt.title("Activation function: sigmoid", fontsize=14)
plt.grid(True)
```



Data Types

(Data Types in Python)

Builtin Data Types

```
dtype             Description
==============================================================
Text Type:        str
Numeric Types:    int, float, complex
Sequence Types:   list, tuple, range
Mapping Type:     dict
Set Types:        set, frozenset
Boolean Type:     bool
Binary Types:     bytes, bytearray, memoryview
None Type:        NoneType
==============================================================
```

Example 1: Getting an int data type ...

```
a = 1
print ( type(a) )
```

Output:

```
< class 'int' >
```

Builtin Data Types

Example 2: Float, complex, boolean, string and list types ...

```
b = 1.5                   # <-- define float ...
print ( type(b) )
c = 1.0 + 1.5j            # <-- define complex ...
print ( type(c) )
d = True                  # <-- define boolean ...
print ( type(d) )
e = "this is a string"    # <-- define string ...
print ( type(e) )
f = ["A, "B", "C", "D"]  # <-- define list ...
print ( type(f) )
```

Output:

```
< class 'float' >
< class 'complex' >
< class 'bool' >
< class 'str' >
< class 'list' >
```

Builtin Data Types

Example 3: Formatting data type output ...

```
print("--- a = {:2d} ... ".format(a) );      # <-- Format integer output.
print("--- b = {:.2f} ... ".format(b) );     # <-- two-decimal places
print('--- c = {:.2f}'.format(c))            #     of accuracy.
print("--- d = {:.5s} ... ".format( str(d) ))
print("--- e = {:15s} ... ".format(e) )
output = ["%.5s" % elem for elem in f ]      # <-- convert list to string ...
print("--- f = ", output )
```

Output:

```
--- a =  1 ...
--- b = 1.50 ...
--- c = 1.00+1.50j
--- d = True ...
--- e = this is a string ...
--- f =  ['A', 'B', 'C', 'D']
```

Floating-Point Numbers

Definition. Floating point variables and constants are used represent values outside of the integer range (e.g., `3.4`, `-45.33` and `2.714`) and are either very large or small in magnitude, (e.g., `3.0e-25`, `4.5e+05`, and `2.34567890098e+19`).

IEEE 754 Floating-Point Standard. Specifies that a floating point number take the form:

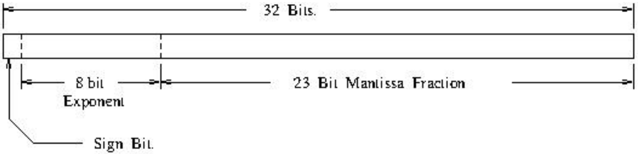
$$X = \sigma \cdot m \cdot 2^{E}. \tag{1}$$

Here:

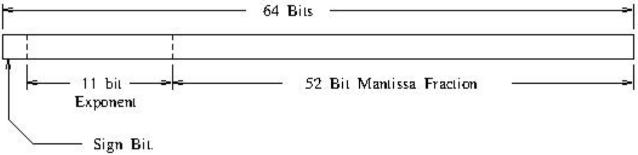
- σ represents the sign of the number.
- m is the mantissa (interpreted as a fraction $0 < m < 1$).
- E is the exponent.

IEEE 754 Floating-Point Standard

Ensures floating point implementions and arithmetic are consistent across various types of computers (e.g., PC and Mac).



IEEE FLOATING POINT ARITHMETIC STANDARD FOR 32 BIT WORDS.



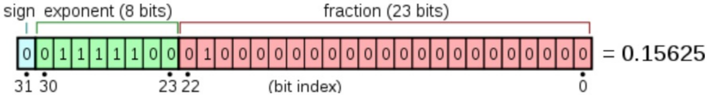
IEEE FLOATING POINT ARITHMETIC STANDARD FOR DOUBLE PRECISION FLOATS.

Largest and Smallest Floating-Point Numbers

```
====================================================================
                  Default
Type   Contains     Value   Size      Range and Precision
====================================================================

float  IEEE 754       0.0   32 bits   +- 13.40282347E+38 /
       floating point                 +- 11.40239846E-45

       Floating point numbers are represented to approximately
       6 to 7 decimal places of accuracy.

double IEEE 754       0.0   64 bits   +- 11.7976931348623157OE+308 /
       floating point                 +- 14.94065645841246544E-324

       Double precision numbers are represented to approximately
       15 to 16 decimal places of accuracy.
====================================================================
```

Working with Double Precision Numbers

Simple Example. Here is the floating point representation for 0.15625



Note. Keep in mind that floating-point numbers are stored in a binary format – this can lead to surprises.

For example, when the decimal fraction 1/10 (0.10 in base 10) is converted to binary, the result is an expansion of infinte length.

Bottom line: You cannot store 0.10 precisely in a computer.

Working with Double Precision Numbers

Accessing the Math Library Module

```
import math;  # <-- import the math library ...
```

Math Constants

```
Method          Description
==========================================================================
math.e          Returns Euler's number (2.7182 ...).

math.inf        Returns floating-point positive infinity.

math.pi         Returns PI (3.1415926 ...).
==========================================================================
```

Math Methods

```
Method          Description
==========================================================================
math.acos()     Returns the arc cosine of a number.
math.acosh()    Returns the inverse hyperbolic cosine of a number.
math.asin()     Returns the arc sine of a number.
math.asinh()    Returns the inverse hyperbolic sine of a number.
==========================================================================
```

Working with Double Precision Numbers

Math Methods (continued) ...

```
Method           Description
====================================================================
math.atan()      Returns the arc tangent of a number in radians
math.atan2()     Returns the arc tangent of y/x in radians
math.ceil()      Rounds a number up to the nearest integer
math.cos()       Returns the cosine of a number
math.cosh()      Returns the hyperbolic cosine of a number
math.exp()       Returns E raised to the power of x
math.fabs()      Returns the absolute value of a number
math.floor()     Rounds a number down to the nearest integer
math.gcd()       Returns the greatest common divisor of two integers
math.isfinite()  Checks whether a number is finite or not
math.isinf()     Checks whether a number is infinite or not
math.isnan()     Checks whether a value is NaN (not a number) or not
math.isqrt()     Rounds a square root number down to the nearest integer
math.ldexp()     Returns the inverse of math.frexp() which is
                 x * (2**i) of the given numbers x and i
math.lgamma()    Returns the log gamma value of x
====================================================================
```

Working with Double Precision Numbers

Math Methods (continued) ...

```
Method              Description
===================================================================
math.log()          Returns the natural logarithm of a number, or the
                    logarithm of number to base.
math.log10()        Returns the base-10 logarithm of x
math.log1p()        Returns the natural logarithm of 1+x
math.log2()         Returns the base-2 logarithm of x
math.perm()         Returns the number of ways to choose k items from n
                    items with order and without repetition
math.pow()          Returns the value of x to the power of y
math.prod()         Returns the product of all the elements in an iterable
math.radians()      Converts a degree value into radians
math.remainder()Returns the closest value that can make numerator
                    completely divisible by the denominator
math.sin()          Returns the sine of a number
math.sinh()         Returns the hyperbolic sine of a number
math.sqrt()         Returns the square root of a number
math.tan()          Returns the tangent of a number
math.tanh()         Returns the hyperbolic tangent of a number
math.trunc()        Returns the truncated integer parts of a number
===================================================================
```

Working with Double Precision Numbers

Example 4: Formatting PI ...

```
import math;      # <-- import math library.
PI = math.pi;     # <-- create user-defined constant.

print("--- PI = {:.2f} ...".format(PI) );  # <-- 2 decimal places.
print("--- PI = {:.15f} ...".format(PI) ); # <-- 15 decimal places.
print("--- PI = {:8.2f} ...".format(PI) ); # <-- 8 characters wide,
                                           #     2 decimal places.
print("--- PI = {:16.12f} ...".format(PI) );# <-- 16 characters wide,
                                            #     12 decimal places.
print("--- PI = {:16.6e} ...".format(PI) ); # <-- exponential format.
```

Output:

```
--- PI = 3.14 ...
--- PI = 3.141592653589793 ...
--- PI =     3.14 ...
--- PI =   3.141592653590 ...
--- PI =     3.141593e+00 ...
```

First Program

(Evaluate and Plot Sigmoid Function)

Problem Desription

Problem Description

In neural network models, the sigmoid function:

$$\sigma(x) = \left[\frac{1}{1+e^{-x}} \right]. \tag{2}$$

serves as an activation. Our first program evaluates and plots $\sigma(x)$ over the range $x \in [-10, 10]$.

Running the Program

From the terminal window, simply type:

```
prompt >> python3 TestSigmoidFunction.py
```

Evaluate and Plot Sigmoid Function

The Python interpreter/compiler will complain if one or more of the required packages (e.g., matplotlib) are missing.

Use pip to install missing Python Packages

The standard package-management system used to install and manage software packages is called pip (or pip3).

Example: And installation is easy!

```
prompt >> pip3 install numpy
prompt >> pip3 install matplotlib
```

To get a list of installed packages:

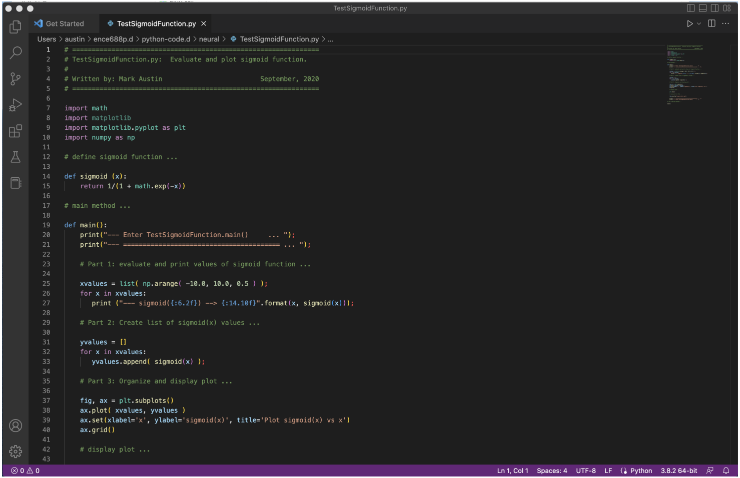
```
prompt >> pip3 list
```

Evaluate and Plot Sigmoid Function

Abbreviated Output:

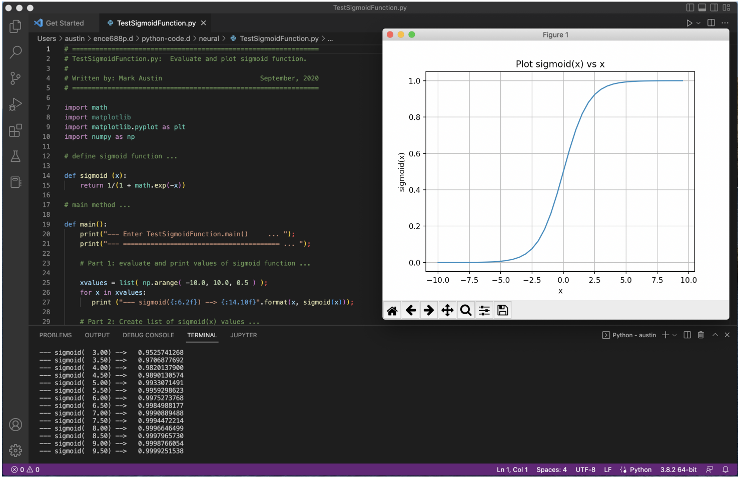
```
Package                Version
---------------------- ---------
....
jupyter                1.0.0
Keras                  2.4.3
....
matplotlib             3.4.1
....
numpy                  1.19.5
....
pandas                 1.1.5
....
scikit-learn           0.24.2
scipy                  1.6.2
....
sklearn                0.0
```

Program Source Code in Visual Studio Code




```python
# ===============================================================
# TestSigmoidFunction.py:  Evaluate and plot sigmoid function.
#
# Written by: Mark Austin                         September, 2020
# ===============================================================

import math
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# define sigmoid function ...

def sigmoid (x):
    return 1/(1 + math.exp(-x))

# main method ...

def main():
    print("--- Enter TestSigmoidFunction.main()     ... ");
    print("--- ======================================== ... ");

    # Part 1: evaluate and print values of sigmoid function ...

    xvalues = list( np.arange( -10.0, 10.0, 0.5 ) );
    for x in xvalues:
        print ("--- sigmoid({:6.2f}) --> {:14.10f}".format(x, sigmoid(x)));

    # Part 2: Create list of sigmoid(x) values ...

    yvalues = []
    for x in xvalues:
        yvalues.append( sigmoid(x) );

    # Part 3: Organize and display plot ...

    fig, ax = plt.subplots()
    ax.plot( xvalues, yvalues )
    ax.set(xlabel='x', ylabel='sigmoid(x)', title='Plot sigmoid(x) vs x')
    ax.grid()

    # display plot ...
```

Program Source Code + Output in Visual Studio Code



Program Source Code

```
# ==========================================================
# TestSigmoidFunction.py: Evaluate/plot sigmoid function.
#
# Written by: Mark Austin                    September, 2020
# ==========================================================

import math
import matplotlib
import matplotlib.pyplot as plt
import numpy as np

# define sigmoid function ...

def sigmoid (x):
    return 1/(1 + math.exp(-x))

# main method ...

def main():
    print("--- Enter TestSigmoidFunction.main() ...");
    print("--- ================================ ...");

    # Part 1: Evaluate and print sigmoid function

    xvalues = list( np.arange( -10.0, 10.0, 0.5 ) );
    for x in xvalues:
       print ("--- sigmoid({:6.2f}) --> {:14.10f}".format(x, sigmoid(x)));

    # Part 2: Create list of sigmoid(x) values ...
```

Program Source Code

```
    # Part 2: Create list of sigmoid(x) values ...

    yvalues = []
    for x in xvalues:
       yvalues.append( sigmoid(x) );

    # Part 3: Organize and display plot ...

    fig, ax = plt.subplots()
    ax.plot( xvalues, yvalues )
    ax.set(xlabel='x', ylabel='sigmoid(x)',
           title='Plot sigmoid(x) vs x')
    ax.grid()

    # display and save plot ...

    plt.show()

    fig.savefig("sigmoid-plot.jpg")

    print("--- ================================ ...");
    print("--- Leave TestSigmoidFunction.main() ...");

# call the main method ...

main()
```

Program Source Code

Points to Note:

- Line comment statements begin with the # character.
- Lines 7-10 import the math, matplotlib, matplotlib.pyplot and numpy modules to the program, and make the functions therein available.
- Functions are the primary method of code organization and reuse in Python.
- User-defined functions are declared with the `def` keyword. A function contains a block of code with an optional `return` keyword.
- Lines 13-14 evaluate and return the sigmoid function.
- Use of the second function, `main()`, is a carry over from programming with C, where all programs begin their execution in `main()`. Its use in Python is optional.

Program Source Code

Points to Note (continued):

- Line 25 creates a list of x coordinates. The numpy function `np.arange()` covers $[-10, 10]$ in increments of 0.5.
- Lines 26-27 systematically traverse the elements of `xvalues`, and compute and print the corresponding values of the `sigmoid()` function.
- Line 27 formats and prints the output. The specification `{:6.2}f` means that the output should be printed as a floating point number, six characters wide, and with two decimal places of accuracy to the right of the decimal point.
- Lines 31-33 traverse the elements of `xvalues`, and systematically assemble a list of sigmoid function `yvalues`.
- Lines 37-47 format a plot of `yvalues` vs `xvalues`, and save to `sigmoid-plot.jpg`.

Builtin Containers and Collections

(Working with Lists, Dictionaries, Sets)

Builtin Containers and Collection

Containers and Collections

A container is an object that stores objects, and provides a way to access and iterate over them. Collections are container data types, namely lists, sets, tuples, dictionary.

Builtin Collection Data Types:

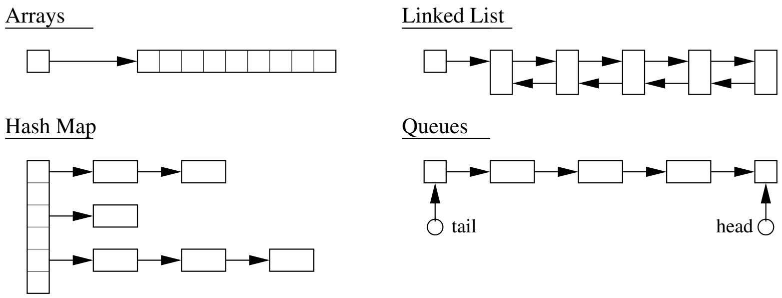
- **List:** A list is a collection which is ordered and changeable.
- **Dictionary:** A dictionary is a collection which is ordered and changeable. No duplicate members.
- **Set:** A set is a collection which is unordered, unchangeable and unindexed. No duplicate members.
- **Tuple:** A tuple is a collection which is ordered and unchangeable.

Working with Lists

List

Lists are used to store multiple items in a single variable. A list may store multiple types (heterogeneous) of elements.

Array, List, HashMap, and Queue Structures



Working with Lists

Basic List Methods

```
Method       Description
==========================================================================
append()     Adds an element at the end of the list
clear()      Removes all the elements from the list
copy()       Returns a copy of the list
count()      Returns the number of elements with the specified value
extend()     Add the elements of a list (or any iterable), to the end of
             the current list.
index()      Returns the index of the first element with the
             specified value.
insert()     Adds an element at the specified position.
remove()     Removes the item with the specified value.
reverse()    Reverses the order of the list.
sort()       Sorts the list.
==========================================================================
```

Working with Lists

Example 1: Create working lists ...

```
  list01 = [ "apple", "orange", "avocado", "banana", "grape", "watermelon"]
  list02 = [ "apple", "avocado", "banana", "banana", "grape", "watermelon"]

  print ("--- List01: %s ..." %( list01 ))
  print ("--- List02: %s ..." %( list02 ))

# Create list with mix of data types ...

  list03 = [ "apple", 40, True, 2.5 ]

  print ("--- List03 (with multiple data types): %s ..." %( list03 ))
```

Output:

```
--- List01: ['apple', 'orange', 'avocado', 'banana', 'grape', 'watermelon'] ...
--- List02: ['apple', 'avocado', 'banana', 'banana', 'grape', 'watermelon'] ...

--- List03 (with multiple data types): ['apple', 40, True, 2.5] ...
```

Working with Lists

Example 2: Access list items ...

```
list04 = list(( "apple", 40, True, 2.5, False ))

print ("---   list04[0]: %s ..." %( list04[0] ))
print ("---   list04[1]: %s ..." %( list04[1] ))
print ("---   list04[2]: %s ..." %( list04[2] ))
print ("---   list04[3]: %s ..." %( list04[3] ))
print ("---   list04[4]: %s ..." %( list04[4] ))
```

Output:

```
---   list04[0]: apple ...
---   list04[1]: 40 ...
---   list04[2]: True ...
---   list04[3]: 2.5 ...
---   list04[4]: False ...
```

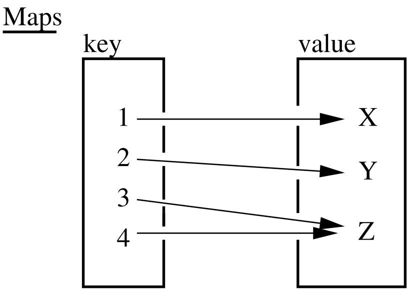
Source Code: See: python-code.d/collections/

Working with Dictionaries

Dictionary

Dictionaries store data values as key:value pairs. As of Python 3.7, a dictionary is a collection which is ordered, changeable and do not allow duplicates.

Key:Value Map Operations



Working with Dictionaries

Basic Dictionary Methods

```
Method        Description
=========================================================================
clear()       Removes all the elements from the dictionary.
copy()        Returns a copy of the dictionary.
fromkeys()    Returns a dictionary with the specified keys and value.
get()         Returns the value of the specified key.
items()       Returns a list containing a tuple for each key value pair.
keys()        Returns a list containing the dictionary's keys.
pop()         Removes the element with the specified key.
popitem()     Removes the last inserted key-value pair.
update()      Updates the dictionary with the specified key-value pairs.
values()      Returns a list of all the values in the dictionary.
=========================================================================
```

Working with Dictionaries

Example 1: Create dictionary of car attributes.

```
car01 = {  "brand": "Honda",         # <-- Create simple dictionary ....
           "model": "Acura",
           "miles": 25000,
             "new": False,
            "year": 2016
        }

print ("--- Car01: %s ..." %( car01 ))  # <-- print dictionary ...
```

Output: Print simple dictionary.

```
--- Car01: {'brand': 'Honda', 'model': 'Acura',
            'miles': 25000, 'new': False, 'year': 2016} ...
```

Working with Dictionaries

Example 2: Systematically access items in Car01 ...

```
print ("--- Car01: brand --> %s ..." %( car01.get("brand") ))
print ("---        : model --> %s ..." %( car01.get("model") ))
print ("---        : miles --> %d ..." %( car01.get("miles") ))
print ("---        : new   --> %s ..." %( car01.get("new") ))
print ("---        : year  --> %d ..." %( car01.get("year") ))
```

Output:

```
--- Access items in Car01 ...
--- Car01: brand --> Honda ...
---      : model --> Acura ...
---      : miles --> 25000 ...
---      : new   --> False ...
---      : year  --> 2016 ...
```

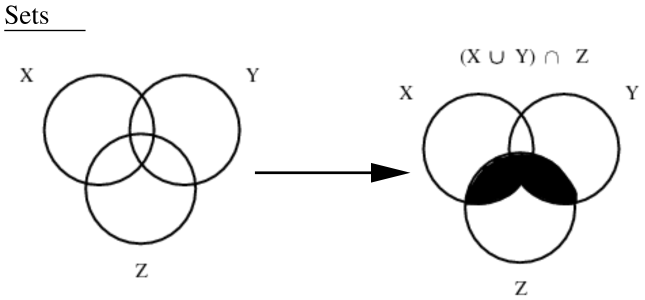
Source Code: See: python-code.d/collections/

Working with Sets

Sets

Sets store multiple items in a single variable. A set is a collection which is unordered, unchangeable (but you can remove items and add new items) and unindexed.

Set Operations



Working with Sets

Basic Set Methods

```
Method         Description
==========================================================================
add()          Adds an element to the set.
clear()        Removes all the elements from the set.
copy()         Returns a copy of the set.
discard()      Remove the specified item.
intersection() Returns a set, that is the intersection of two other sets.
remove()       Removes the specified element.
union()        Return a set containing the union of sets
update()       Update the set with the union of this set and others.
==========================================================================
```

Working with Sets

Example 1: Create working sets; set operations ...

```
set01 = { 1, 2, 3, 4, 5, 6, 7 }
set02 = { 6, 7, 8, 9, 10 }
set03 = {"apple", "banana", "cherry"}
set04 = {True, False, False}

print ("--- Set01.union(Set02) : %s ..." %( set01.union(set02) ))
print ("--- Set01.intersection(Set02) : %s ..."
                                       %( set01.intersection(set02) ))
```

Output:

```
--- Create working sets ...
--- Set01: {1, 2, 3, 4, 5, 6, 7} ...
--- Set02: {6, 7, 8, 9, 10} ...
--- Set03: {'cherry', 'banana', 'apple'} ...
--- Set04: {False, True} ...

--- Set01.union(Set02) : {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} ...
--- Set01.intersection(Set02) : {6, 7} ...
```

Working with Sets

Example 2: Add items to set03, then print ...

```
set03.add("strawberry")
set03.add("kiwi")
print ("--- Set03 (appended): ...")
for x in set03:
    print ("---   %s ..." %(x))
```

Output: Set03 appended ...

```
---   cherry ...
---   strawberry ...
---   banana ...
---   kiwi ...
---   apple ...
```

Source Code: See: python-code.d/collections/

Numerical Python

(NumPy)

Numerical Python (NumPy)

Introduction to NumPy

Numerical Python (NumPy) is an open source Python library that contains computational support for n-dimensional array objects, along with mathematical methods to operate on them.

Key Features:

- Create 0-d, 1-d and 2-d arrays. 3-d blocks.
- Operations on array elements (e.g., min, max, sort).
- Operations on arrays (e.g., reshape, stack).
- Compute matrix properties. Solve matrix equations.

Installation

```
prompt >> pip3 install numpy
```

Numerical Data Types in NumpPy

dtype	Variants	Description
int	int8, int16, int32, int64	Integers
uint	uint8, uint16, uint32, uint64	Unsigned integers
bool	bool	Boolean (True or False)
float	float16, float32, float64, float128	Floating-point numbers
complex	complex64, complex128, complex256	Complex-valued floating point numbers

Working with NumPy

Example 1: Create 0-d, 1-d, and 2-d arrays ...

```
a = np.array(101);   # <-- create 0-d array.
print (a)

a = np.array( [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] ); # <-- create 1-d array of
print (a)

a = np.array( ["The", "Brown", "Fox"] ); # <-- array of charcter strings.
a = np.append(a, "!")

for i in a:                              # <-- loop over array indices ...
   print(i)
```

Output:

```
101
[ 1  2  3  4  5  6  7  8  9 10]
The
Brown
Fox
!
```

Working with NumPy

Example 2: Print each array element and its index ...

```
# Create array of character strings ...

a = np.array( ["The", "Brown", "Fox", "!"] );

for i,e in enumerate(a):
    print("--- Index: {}, was: {}".format(i, e))
```

Output:

```
--- Index: 0, was: The
--- Index: 1, was: Quick
--- Index: 2, was: Brown
--- Index: 3, was: Fox
--- Index: 4, was: !
```

Working with NumPy

Example 3: Sort array elements ...

```
# Sort array of floating point numbers ...

a = np.array( [ 2.3, 1.0, 4.5, -13.0, 100.0, 43, -15.0, 0.0 ] )
print(a);
print(np.sort(a));

# Sort array of state abbreviations ...

a = np.array( ["MD", "CA", "RI", "UT", "LA", "AL", "WA", "OR", "CO"] )
print(a);
print(np.sort(a))
```

Output:

```
--- Sort array of floating-point numbers ...
[  2.3    1.     4.5  -13.   100.    43.   -15.     0. ]
[-15.   -13.     0.     1.     2.3    4.5   43.   100. ]
--- Sort array of state abbreviations ...
['MD' 'CA' 'RI' 'UT' 'LA' 'AL' 'WA' 'OR' 'CO']
['AL' 'CA' 'CO' 'LA' 'MD' 'OR' 'RI' 'UT' 'WA']
```

Working with NumPy

Example 4: Create two-dimensional array ...

```
c = np.array( [ ( 0, 1, 4, 3, 2), ( 3, 4, 5, 6, 7),
                ( 6, 7, 8, 9,10), ( 9,10,11,12,13) ] );

PrintMatrix("C", c);        # <-- print formatted matrix ....

print("     Min: {}".format(np.min(c)))
print("     Max: {}".format(np.max(c)))
print("     Average: {}".format(np.average(c)))
print("     Max array index: {}".format(np.argmax(c)))
```

Output:

```
Matrix: C
   0.000    1.000    4.000    3.000    2.000
   3.000    4.000    5.000    6.000    7.000
   6.000    7.000    8.000    9.000   10.000
   9.000   10.000   11.000   12.000   13.000

Min: 0                  Average: 6.5
Max: 13                 Max array index: 19
```

Working with NumPy

Example 5: Create three-dimensional array block ...

```
c = np.array( [ [ ( 0, 1), (3, 4) ], [(5, 6), (7, 8) ] ] );
print(c)
```

Output:

```
[ [ [0 1]
    [3 4] ]

  [ [5 6]
    [7 8] ] ]
```

Working with NumPy

Example 6: Reshape 1-d array $\longrightarrow$ 2-d matrix ...

```
d1 = np.arange(20);      # <-- create 1-d test array ...
print(d1);

d1 = d1.reshape(4,5);    # <-- reshape to (4x5) array ...
PrintMatrix("(4x5)", d1 );
```

Output:

```
--- 1-d test array:

  [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]

--- Reshape to (4x5) matrix ...

Matrix: (4x5)
    0.000    1.000    2.000    3.000    4.000
    5.000    6.000    7.000    8.000    9.000
   10.000   11.000   12.000   13.000   14.000
   15.000   16.000   17.000   18.000   19.000
```

Working with NumPy

Example 7: Create horizontal and vertical array stacks ...

```
d1 = np.array( [ ( 0, 1), ( 3, 4) ] );    # <-- create test arrays ...
d2 = np.array( [ ( 5, 6), ( 7, 8) ] );

PrintMatrix("d1", d1 ); PrintMatrix("d2", d2 );

h1 = np.hstack((d1, d2));                  # <-- create horizontal stack ...
PrintMatrix( "np.hstack(d1, d2)", h1 );
h2 = np.vstack((d1, d2));                  # <-- create vertical stack ...
PrintMatrix( "np.vstack(d1, d2)", h2 );
```

Output:

```
Matrix: d1                    Matrix: np.hstack(d1, d2)
   0.000     1.000               0.000     1.000     5.000     6.000
   3.000     4.000               3.000     4.000     7.000     8.000

Matrix: d2                    Matrix: np.vstack(d1, d2)
   5.000     6.000               0.000     1.000
   7.000     8.000               3.000     4.000
                                 5.000     6.000
                                 7.000     8.000
```

Working with NumPy

Example 8: Exercise np.zeros() and np.eye() ...

```
matrix02 = np.zeros(shape=(4, 4))   # <-- create (4x4) array of zeros.
PrintMatrix("matrix02", matrix02 );

matrix03 = np.eye(4, dtype = float) # <-- create (4x4) identidy matrix.
PrintMatrix("matrix03", matrix03 );
```

Output:

```
Matrix: matrix02
   0.000    0.000    0.000    0.000
   0.000    0.000    0.000    0.000
   0.000    0.000    0.000    0.000
   0.000    0.000    0.000    0.000

Matrix: matrix03
   1.000    0.000    0.000    0.000
   0.000    1.000    0.000    0.000
   0.000    0.000    1.000    0.000
   0.000    0.000    0.000    1.000
```

Working with NumPy

Example 9: Reshape arrays of random numbers

```
matrix06 = np.random.random((20,1));   # <-- create (20x1) array
PrintMatrix("matrix06", matrix06 );    #     of random numbers.

PrintMatrix ( "matrix06 (reshaped)",   # <-- reshape to (10x2).
              matrix06.reshape(10,2) )
```

Abbreviated Output:

```
--- Original (20x1) matrix     --- Reshape to (10x2) matrix ...

Matrix: matrix06               Matrix: matrix06 (reshaped)
   0.326                          0.326    0.459
   0.459                          0.545    0.419
   0.545                          0.537    0.632
   .....                          .....    .....
   0.803                          .....    .....
   0.014                          0.165    0.803
   0.291                          0.014    0.291
```

Working with NumPy

Example 10: Solve the family of matrix equations:

$$\begin{bmatrix} 3 & -6 & 7 \\ 9 & 0 & -5 \\ 5 & -8 & 6 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 3 \\ -4 \end{bmatrix} \tag{3}$$

Part I: Theoretical Considerations:

- A unique solution $\{X\} = [A^{-1}] \cdot \{B\}$ exists when $[A^{-1}]$ exists (i.e., det $[A] \neq 0$). Expanding det(A) about the first row gives:

$$\begin{aligned} det(A) &= 3\text{det}\begin{bmatrix} 0 & -5 \\ -8 & 6 \end{bmatrix} + 6\text{det}\begin{bmatrix} 9 & -5 \\ 5 & 6 \end{bmatrix} + 7\text{det}\begin{bmatrix} 9 & 0 \\ 5 & -8 \end{bmatrix}, \\ &= 3(0-40) + 6(54+25) + 7(-72-0) = -150. \end{aligned} \tag{4}$$

Working with NumPy

Part II: Program Source Code:

```
# ================================================================
# TestMatrixEquations01.py: Compute solution to matrix equations.
#
# Written by: Mark Austin                           November 2022
# ================================================================

import numpy as np
from numpy.linalg import matrix_rank

# Function to print two-dimensional matrices ...

def PrintMatrix(name, a):
    print("Matrix: {:s} ".format(name) );
    for row in a:
        for col in row:
            print("{:8.3f}".format(col), end=" ")
        print("")

# main method ...

def main():
    print("--- Enter TestMatrixEquations01.main()        ... ");
    print("--- ======================================== ... ");

    print("--- Part 1: Create test matrices ... ");
```

Working with NumPy

Part II: Program Source Code: (Continued) ...

```
    A = np.array( [ [ 3, -6,  7],
                    [ 9,  0, -5],
                    [ 5, -8,  6] ]);
    PrintMatrix("A", A);

    B = np.array([ [3], [3], [-4] ]);
    PrintMatrix("B", B);

    print("--- Part 2: Check properties of matrix A ... ");

    rank = matrix_rank(A)
    det  = np.linalg.det(A)

    print("--- Matrix A: rank = {:f}, det = {:f}  ...".format(rank, det) );

    print("--- Part 3: Solve A.x = B ... ");

    x = np.linalg.solve(A, B)
    PrintMatrix("x", x);

    print("--- ======================================== ... ");
    print("--- Leave TestMatrixEquations01.main()      ... ");

# call the main method ...

main()
```

Working with NumPy

Part III: Program Output:

```
# Part 1: Create test matrices ...

Matrix: A
   3.000   -6.000    7.000
   9.000    0.000   -5.000
   5.000   -8.000    6.000

Matrix: B
   3.000
   3.000
  -4.000

# Part 2: Check properties of matrix A ...

Matrix A: rank = 3.000000, det = -150.000000  ...
```

```
# Part 3: Solve A.x = B ...

Matrix: x
   2.000
   4.000
   3.000
```

Data and

Dataset Transformation

(Pandas)

Working with Pandas

Introduction to Pandas

Pandas is an open source Python Library that supports working and analysis of tabular data sets.

Benefits:

- Pandas can clean messy data sets, and make them readable and relevant.
- Pandas allows us to analyze large data sets and make conclusions based on statistical theories.
- Three data structures: Series, DataFrame and Panel.

Installation:

```
prompt >> pip3 install pandas
```

What can Pandas do?

Basic Operations:

- Create series and dataframes.
- Read CSV and JSON files.
- Plot data.

Clean Data:

- Clean empty cells.
- Fix wrong format.
- Remove duplicates.

Advanced Operations:

- Combine (concatenate, join, merge) Panda objects.
- Compute correlations.

Panda Series and DataFrames

Panda Series

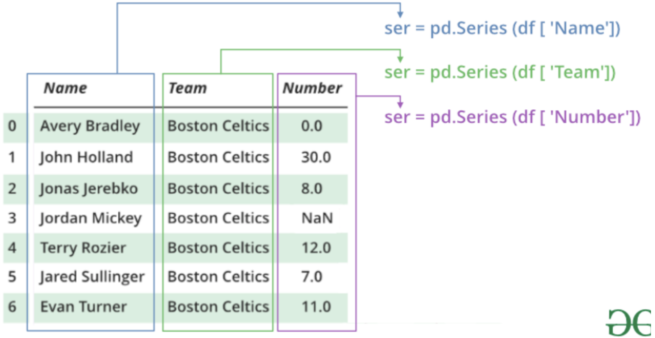
A Panda Series is a one-dimensional ... labeled array capable of holding data of any type (integer, string, float, python objects, etc.).

Panda DataFrame

A Panda DataFrame is a two-dimensional (potentially heterogeneous) tabular data structure with labeled axes for the rows and columns.

Panda Series

Panda Series Elements: columns, data ...



	Name	*Team*	*Number*
0	Avery Bradley	Boston Celtics	0.0
1	John Holland	Boston Celtics	30.0
2	Jonas Jerebko	Boston Celtics	8.0
3	Jordan Mickey	Boston Celtics	NaN
4	Terry Rozier	Boston Celtics	12.0
5	Jared Sullinger	Boston Celtics	7.0
6	Evan Turner	Boston Celtics	11.0

Basic Operations:

- Create a series; access elements; index and select data; binary operations; conversion operations.

Panda Series

Example 1: Manually create series from list:

```
# Part 1: Manually create series ...

a = [1, 2, 3, 4, 3, 2, 1 ]
myvar = pd.Series(a)
print(myvar)

# Part 2: Create series from a list with labels ...

myvar = pd.Series(a, index = ["a", "b", "c", "d", "c", "b", "a" ])
print(myvar)
```

Abbreviated Output: Parts 1 and 2 ...

```
Part 01                 Part 02
0     1                 a     1
1     2                 b     2
.....                   .....
5     2                 b     2
6     1                 a     1
dtype: int64            dtype: int64
```

Panda Series

Example 2: Manually create series from dictionary:

```
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

Output:

```
day1    420
day2    380
day3    390
dtype: int64
```

Panda Series

Example 3: Create series from NumPy functions

```
# series01 = pd.Series(np.arange(2,8)) ... ");

series01 = pd.Series(np.arange(2,8))
print(series01)
```

Output:

```
0    2
1    3
2    4
3    5
4    6
5    7
dtype: int64
```

Panda Series

Example 4: Create series from NumPy functions

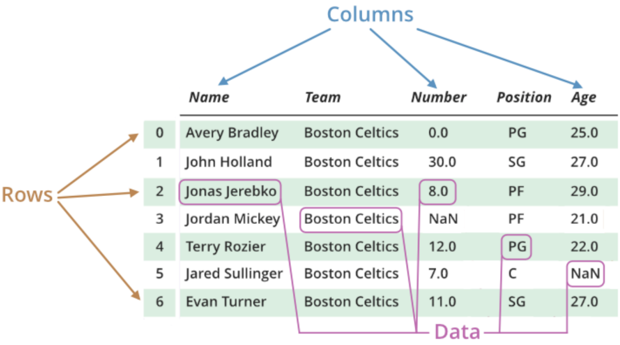
```
series02 = pd.Series( np.linspace(0,10,5) )
print(series02)

print( series02.size)
print( len(series02) )
print( series02.values )
```

Output:

```
0     0.0
1     2.5
2     5.0
3     7.5
4    10.0
dtype: float64

5                          # <-- series02.size ...
5                          # <-- series02 length ...
[ 0.   2.5  5.   7.5 10. ] # <-- series02 values ...
```

Panda DataFrames

Panda DataFrame Elements: rows, columns, data ...



	Name	*Team*	*Number*	*Position*	*Age*
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0

Basic Operations:

- Create dataframe; deal with rows and columns; index and select data; iterate over rows and columns.

Working with Panda DataFrames

Example 1: Manually create small dataset ...

```
mydataset = {
   'cars': [ "BMW", "Honda", "Acura"],
   'year': [ 2013,     2017,    2022]
}

myvar = pd.DataFrame(mydataset)
print(myvar)
```

Output:

```
      cars  year
0     BMW  2013
1   Honda  2017
2   Acura  2022
```

Working with Panda DataFrames

Example 2: Create dataframes from 1-d and 2-d arrays ...

```
myvar = pd.DataFrame( np.arange(1,8) ) # <-- dataframe from 1-d array
print(myvar)

df = pd.DataFrame( [ [1,2],
                     [3,4],
                     [5,6] ] )         # <-- dataframe from 2-d array
print(df)
```

Abbreviated Output:

```
Dataframe from 1-d np array      Dataframe from 2-d np array
---------------------------      ---------------------------
   0                                    0  1
0  1                                 0  1  2
1  2                                 1  3  4
2  3                                 2  5  6
....
5  6
6  7
```

Working with Panda DataFrames

Example 3: Create simple dataframe from multiple series ...

```
data = {                           # <-- Create dataframe from
   "calories": [520, 480, 400],    #     multiple series.
   "duration": [ 50,  48,  40]
}

myvar = pd.DataFrame(data)
print(myvar)

index = ["day1", "day2", "day3"]  # <-- give each ros a new name.
myvar = pd.DataFrame(data, index)
print(myvar)
```

Output:

```
Part 1: dataframe from series        Part 2: rename rows
     calories  duration                      calories  duration
 0        520        50               day1        520        50
 1        480        48               day2        480        48
 2        400        40               day3        400        40
```

Working with Panda DataFrames

Example 4: Create dataframe from JSON object ...

```
# Create JSON object (same format as Python dictionary) ...

data = {
  "Duration":{ "0":60,  "1":60, "2":60,   "3":45,  "4":45,  "5":60 },
  "Pulse":{    "0":110, "1":117, "2":103, "3":109, "4":117, "5":102 },
  "Maxpulse":{ "0":130, "1":145, "2":135, "3":175, "4":148, "5":127 },
  "Calories":{ "0":409, "1":479, "2":340, "3":282, "4":406, "5":300 }
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
   Duration  Pulse  Maxpulse  Calories
0        60    110       130       409
1        60    117       145       479
2        60    103       135       340
3        45    109       175       282
4        45    117       148       406
5        60    102       127       300
```

Working with Panda DataFrames

Example 5: Select rows and columns from dataframe ...

```
# Select columns of a dataframe ...

print( df[ [ 'Duration','Calories'] ].head() )

# Selecting rows of a dataframe ...

print( df.loc['1'].head() )     # <-- extract and print row 1
print( df.loc['2'].head() )     # <-- extract and print row 2
```

Output:

```
Columns of dataframe        Row 1                    Row 2
--------------------        -----------------        -----------------

   Duration  Calories       Duration      60         Duration      60
0        60       409       Pulse        117         Pulse        103
1        60       479       Maxpulse     145         Maxpulse     135
2        60       340       Calories     479         Calories     340
3        45       282       Name: 1, dtype: int64    Name: 2, dtype: int64
4        45       406
```

Working with Pandas

Example 6: Read and plot CSV data file.

```
df = pd.read_csv('../data/AirPassengers.csv')
print(df.head())

print(df.info())  # <-- print dataframe info and shape ...
print(df.shape)
```

Output:

```
     Month  #Passengers
0  1949-01          112
1  1949-02          118
2  1949-03          132
3  1949-04          129
4  1949-05          121
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 2 columns):
#   Column       Non-Null Count  Dtype
---  ------       --------------  -----
0   Month        144 non-null    object
1   #Passengers  144 non-null    int64
dtypes: int64(1), object(1)
memory usage: 2.4+ KB
None
(144, 2)
```

Working with Pandas

Example 6: (continued)

```
import matplotlib.pyplot as plt

ax = plt.gca()
df.plot(kind='line',x='Month',y='#Passengers',color='blue',ax=ax)
plt.show()
```

Output: