SBT 645 Introduction to Scientific Computing in Sports Science

De Motu Animalium G.Borelli (1680)

#2

SERDAR ARITAN

serdar.aritan@hacettepe.edu.tr

Biyomekanik Araştırma Grubu
www.biomech.hacettepe.edu.tr
Spor Bilimleri Fakültesi
www.sbt.hacettepe.edu.tr
Hacettepe Universitesi, Ankara, Türkiye
www.hacettepe.edu.tr

Download & Install WinPython

Download & Install Anaconda

Anaconda Installers

Windows

Python 3.8

64-Bit Graphical Installer (457 MB)

32-Bit Graphical Installer (403 MB)

MacOS

Python 3.8

64-Bit Graphical Installer (435 MB)

64-Bit Command Line Installer (428 MB)

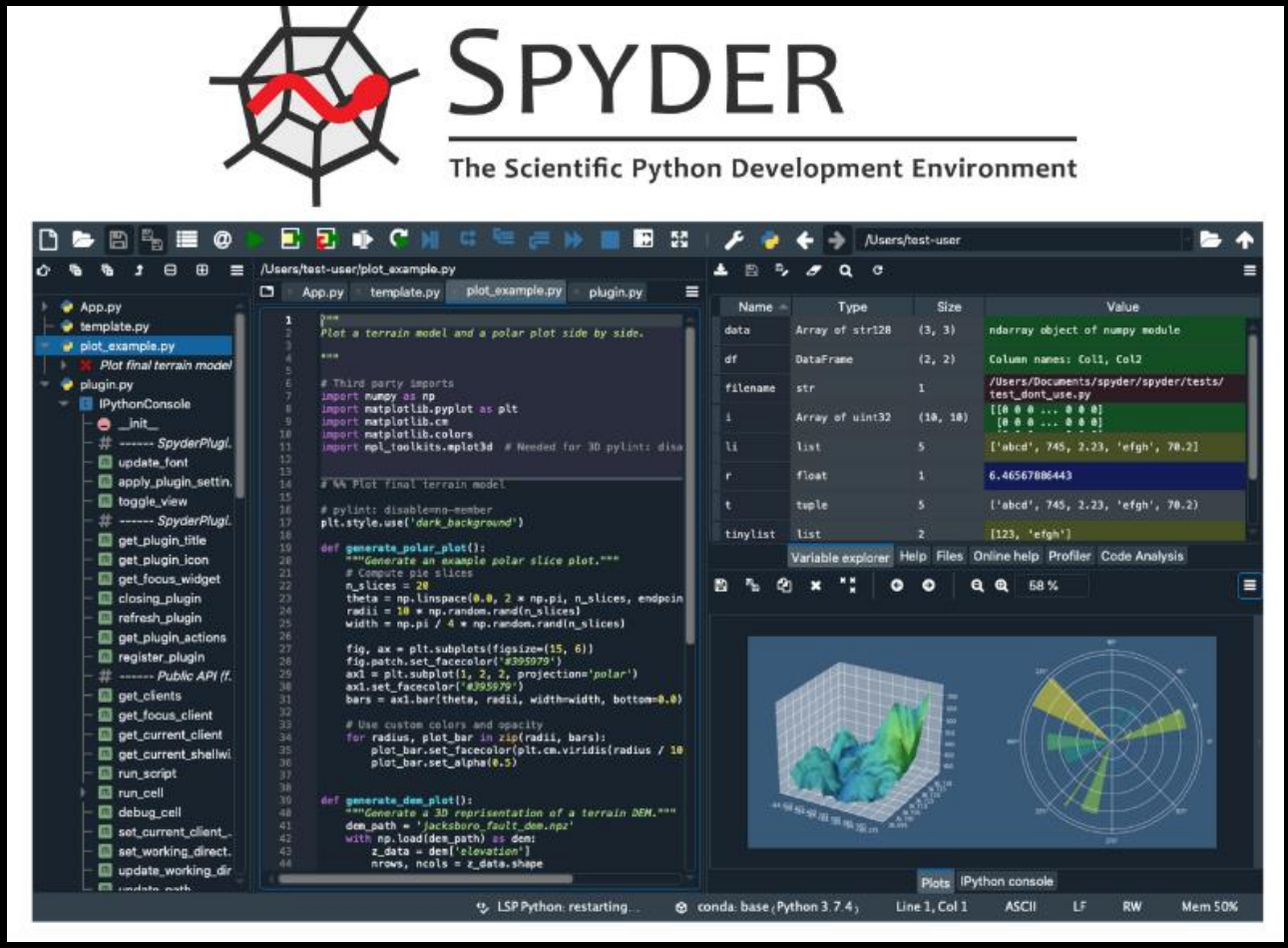
Linux

Python 3.8

64-Bit (x86) Installer (529 MB)

64-Bit (Power8 and Power9) Installer (279 MB)

Download & Install Spyder



How do I get the computer to solve a problem?

- **The algorithm has to be programmed.**
  The single steps to be executed (like a recipe) have to be described in a language which the computer understands.

- **The exists many programming languages e.g. FORTRAN, Algol, BASIC, Java, C, C++, C#, Ada, A#, Pascal, Matlab, Scilab, Python, Oberon, Eiffel, Maple, Mathematica . . .**

- **For each such language there exist compilers which translate the program in executable machine code for the specific computer.**

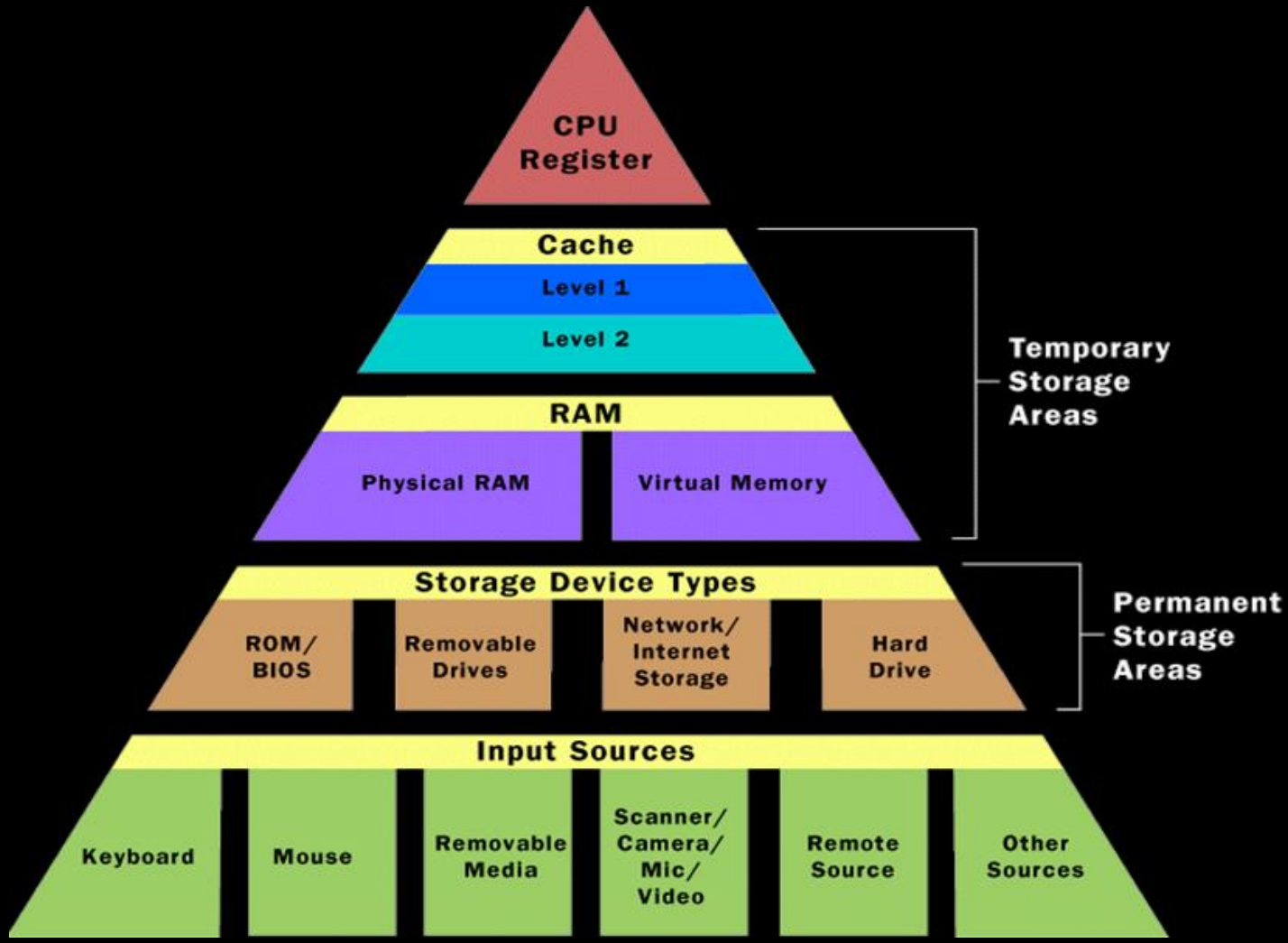
- **I will use in my examples Matlab and Python. However, the language is not relevant for the goal to hopefully arouse your enthusiasm for programming!**

Computer Memory

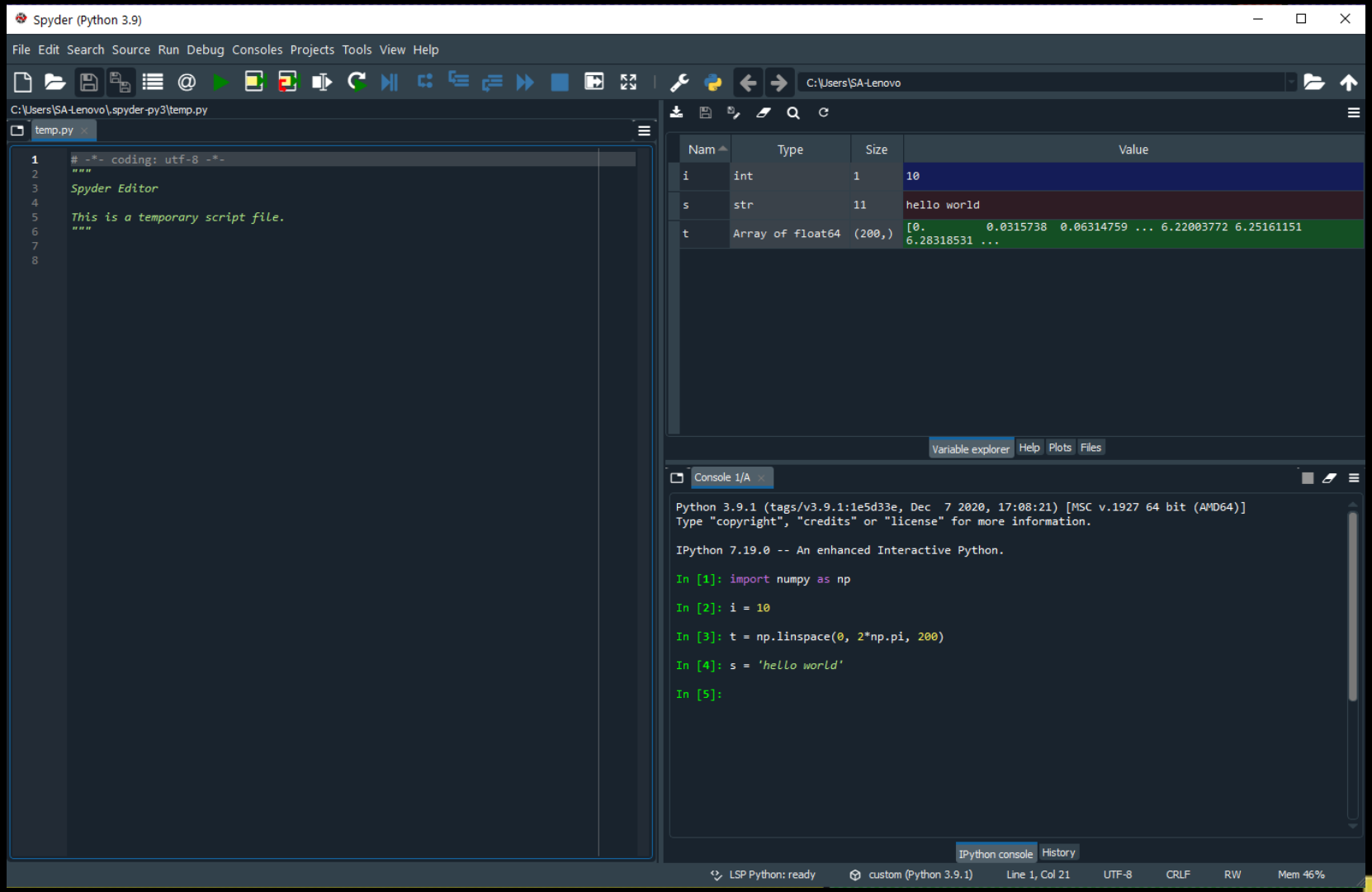
- Numbers and the results of numeric computations (along with other data such as text, graphics, documents, etc) must be stored somewhere in a computer.

- That “somewhere” is “memory”.
- Memory comes in a variety of types and speeds:
- Cache – in the CPU itself (fastest)
- RAM - external to the CPU (fast)
- Disk - physical media, external to the CPU, r/w
- CDROM - physical media, (slow)
- Tape - physical media, (slowest)
- Memory is measured in “bytes” (and kilobytes, megabytes, gigabytes, and terabytes.)

Computer Memory is Varied

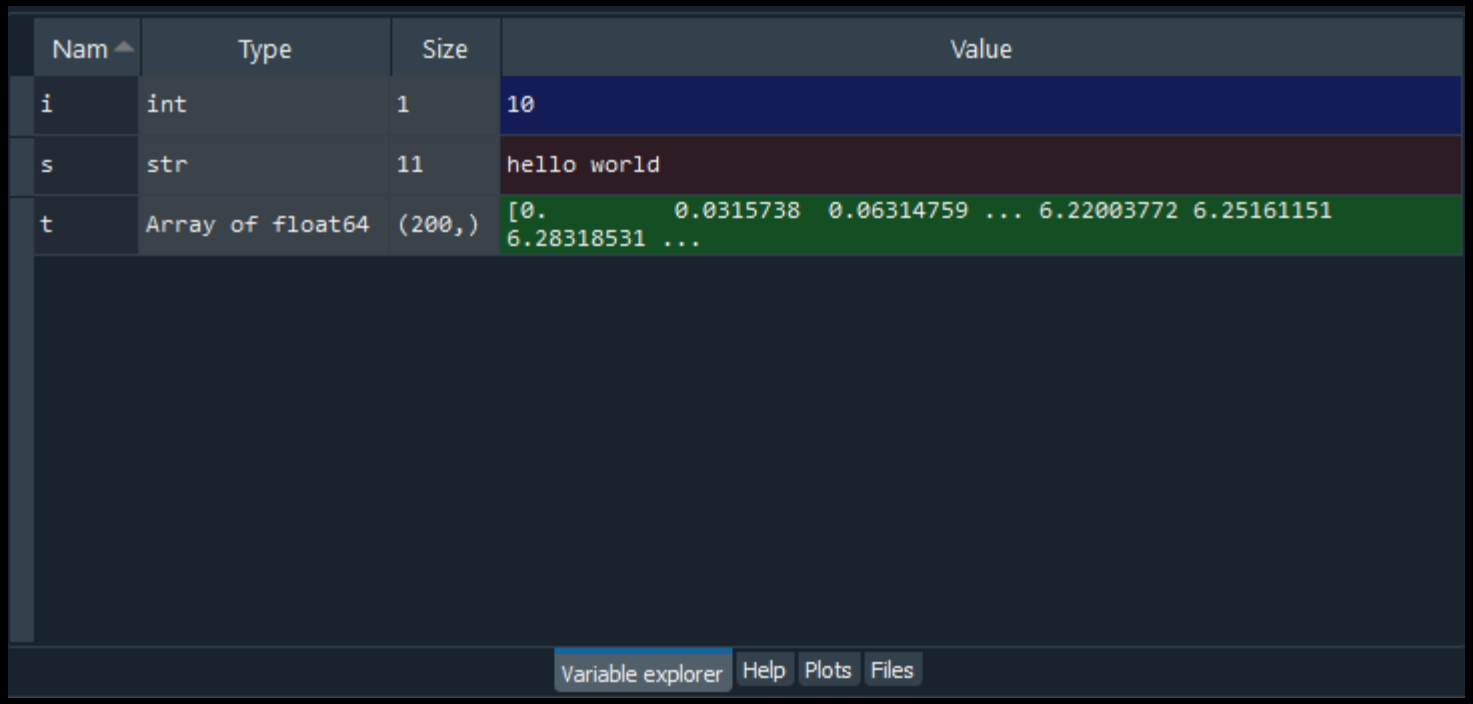


Memory in Python



Inside the Bytes

- In the previous slide, we see:



Nam	Type	Size	Value
i	int	1	10
s	str	11	hello world
t	Array of float64	(200,)	[0. 0.0315738 0.06314759 ... 6.22003772 6.25161151 6.28318531 ...

Variable explorer | Help | Plots | Files

- What is the size of the variable “i”
- What does “type” represent?
- How many bytes are used to store the value?

- Numbers we use are DECIMAL (or base 10):
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - $123 = 1*10^2 + 2*10^1 + 3*10^0$
- But we can always use other bases:
- Octal (base 8):
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7
  - $123 = 1*8^2 + 2*8^1 + 3*8^0$
  - $123_8 = 64+16+3 = 83_{10}$
- Binary (base 2):
  - Digits: 0, 1
  - $1011 = 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0$
  - $1011_2 = 8+0+2+1 = 11_{10}$
  - $123_8 = 001\ \ 010\ \ 011 = 1010011_2$
- Hexidecimal (base 16):
  - Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - $123 = 1*16^2 + 2*16^1 + 3*16^0$
  - $123_{16} = 256 + 32 + 3 = 291_{10}$
  - $123_{16} = 0001\ \ 0010\ \ 0011 = 100100011_2$

Inside the Bytes

- A byte is the smallest memory allocation available.
- A byte contains 8 bits so that:
  - *Smallest: 0 0 0 0 0 0 0 0 = 0_{10}*
  - *Largest: 1 1 1 1 1 1 1 1 = $1*2^7+1*2^6+1*2^5+1*2^4+1*2^3+1*2^2+1*2^1+1*2^0$ = 255_{10} (or 2^8-1)*
- Result: a single byte can be used to store an integer number ranging from 0 to 255 (256 different numbers)
- If negative numbers are included, one bit must be dedicated to the sign, leaving only 7 bits for the number
  - *Smallest: 0*
  - *Largest: $+127_{10}$ or -128_{10}*

Inside the Bytes

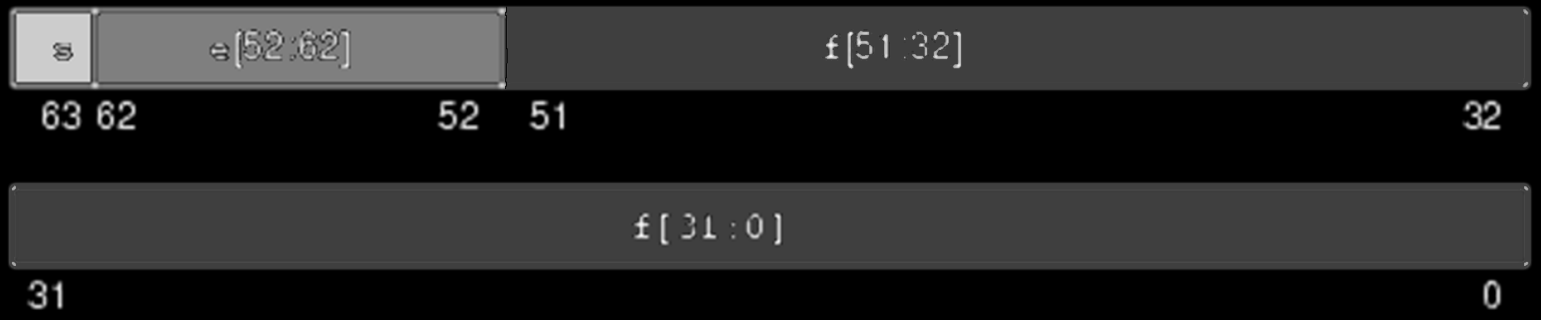
- 2^8-1 (255) is not a very big number, so computers generally use multiple bytes to represent numbers:

- Here is some vocabulary:
  - byte = 8 bits
  - single (precision) = 4 bytes
  - double (precision) = 8 bytes
  - quad (precision) = 16 bytes

  - char = 2 bytes (used to be 1 byte)

Inside the Bytes

- How are fractional (floating point) numbers stored in a computer?
- The IEEE 754 double format consists of three fields:
  - *a 52-bit fraction, f*
  - *an 11-bit biased exponent, e*
  - *and a 1-bit sign, s*
- These fields are stored contiguously in 8 bytes (or 2 successively addressed 4-byte words):



Inside the Bytes

- Because only a finite number of bits can be used for each number, not all possible numbers can be represented:
- Negative numbers less than $-(2-2^{-52}) \times 2^{1023}$
  (negative overflow)
- Negative numbers greater than -2^{-1022}
  (negative underflow)
- Positive numbers less than 2^{-1022}
  (positive underflow)
- Positive numbers greater than $(2-2^{-52}) \times 2^{1023}$
  (positive overflow)
- Zero (actually is a special combination of bits)

Inside the Bytes

- Others sources of error in computation:
- Errors in the input data - measurement errors, errors introduced by the conversion of decimal data to binary, roundoff errors.
- Roundoff errors during computation (as discussed)
- Truncation errors - using approximate calculation is inevitable when computing quantities involving limits and other infinite processes on a computer
  - Never try to compare two floating point numbers for equality because all 16 digits would have to match perfectly…

Inside the Bytes

- Precision
- The smallest difference that can be represented on the computer
- Accuracy
- How close your answer is to the “actual” or “real” answer.
- Recognize:
- Python that use IEEE doubles give you 15-16 “good” digits

Machine Epsilon

A fundamental quantity for any floating point system is the unit round off. This is denoted by u and is the smallest positive number where the computed value of 1 + u is different from 1. The **machine epsilon** is the smallest a − 1 where a is the smallest representable number greater than 1.

Machine Epsilon

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308,
max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)

>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

Machine Epsilon

Variables are chunks of data stored in the computers memory. There are generally three types of data stored in variables. Variables can be in the form of integers or in a string. The second type of data, called a float, refers to the non-whole numbers like decimals.

integers 1, 2, 3, 4, ...

float 0.3, 1.1, 1.8, 2.5, 3.14, ...

string “what is your name ? “, “your age is 18”, ...

Machine Epsilon

```
>>>  0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17

# try these
>>> 1.0023 - 1.0567


>>> 1000.0023 - 1000.0567
```

Machine Epsilon

```
# However, with decimals, the result can be exact:
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -
Decimal('0.3')
Decimal('0.0')
# When decimals of different precision are mixed in
expressions,
# Python converts up to the largest number of decimal digits
>>> Decimal('0.1')+Decimal('0.10')+Decimal('0.10')-
Decimal('0.30')
Decimal('0.00')
```

The decimal object, formally known as Decimal. Syntactically, you create decimals by calling a function within an imported module, rather than running a literal expression. Functionally, decimals are like floating-point numbers, but they have a fixed number of decimal points. Hence, decimals are fixed-precision floating-point values. For example, with decimals, we can have a floating-point value that always retains just two decimal digits.

Variables

- A variable is a placeholder in memory
- Variables contain values
- Variable names:
  - Are case sensitive: **Cost**, **cost**, **COST** are different
  - May contain up to **31** characters (more are ignored)
  - Must start with a letter,
  - May contain numbers and letters
  - May NOT contain punctuation except "_"

Variables

MIXED TYPES ARE CONVERTED UP

integers are simpler than floating point numbers, which are simpler than complex numbers. So, when an integer is mixed with a floating point, as in the preceding example, the integer is converted up to a floating-point value first, and floating-point math yields the floating-point result.

```
>>> 40 + 3.14          # Integer to float, float math/result
43.14
```

You can force the issue by calling built-in functions to convert types manually

```
>>> int(3.1415)        # Truncates float to integer
3
>>> float(3)           # Converts integer to float
3.0
```

Keywords

Do NOT use these words as a variable name

```
>>>help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get
more help.

False               break               for                 not
None                class               from                or
True                continue            global              pass
__peg_parser__      def                 if                  raise
and                 del                 import
return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
```

Variables and Assignment

Variables are named locations in memory where numbers, strings and other elements of data may be stored while the program is working. Variable names are combinations of letters and digits, but must start with a letter.

```
variable = expression;
```

for example

```
>>> a = 6
>>> b = 4
>>> c = a + b
>>> a = b # what is a To see try print(a)
```

or

```
>>> name = 'Mark'
>>> fruit = 'Apple'
>>> fruit = name # ???
```

To dispay contents of a variable, use

```
print(variable)
>>> print(fruit) # ???
```

NumPy

The fundamental package for scientific computing with Python

NumPy is available at www.numpy.org.

Nearly every scientist working in Python draws on the power of NumPy.

NumPy brings the computational power of languages like C and Fortran to Python, a language much easier to learn and use. With this power comes simplicity: a solution in NumPy is often clear and elegant.

Quantum Computing	Statistical Computing	Signal Processing	Image Processing	Graphs and Networks	Astronomy Processes	Cognitive Psychology
QuTiP PyQuil Qiskit	Pandas statsmodels Seaborn	SciPy PyWavelets	Scikit-image OpenCV	NetworkX graph-tool igraph PyGSP	AstroPy SunPy SpacePy	PsychoPy

Bioinformatics	Bayesian Inference	Mathematical Analysis	Simulation Modeling	Multi-variate Analysis	Geographic Processing	Interactive Computing
BioPython Scikit-Bio PyEnsembl	PyStan PyMC3	SciPy SymPy cvxpy FEniCS	PyDSTool	PyChem	Shapely GeoPandas Folium	Jupyter IPython Binder

The NumPy Array Object

Vectors, matrices, and arrays of higher dimensions are essential tools in numerical computing. When a computation must be repeated for a set of input values, it is natural and advantageous to represent the data as arrays and the computation in terms of array operations. Computations that are formulated this way are said to be vectorized. Vectorized computing eliminates the need for many explicit loops over the array elements by applying batch operations on the array data. The result is concise and more maintainable code, and it enables delegating the implementation of (e.g., elementwise) array operations to more efficient low-level libraries. Vectorized computations can therefore be significantly faster than sequential element-by-element computations.

For example:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
```

The NumPy Array Object

2-D, or two-dimensional array, and so on. The NumPy `ndarray` class is used to represent both matrices and vectors. A vector is an array with a single dimension (there's no difference between row and column vectors), while a matrix refers to an array with two dimensions. For 3-D or higher dimensional arrays, the term tensor is also commonly used.

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
```

We can access the elements in the array using square brackets. When you're accessing elements, remember that indexing in NumPy starts at 0.

The NumPy Array Object

The core of the NumPy library is the data structures for representing multidimensional arrays of homogeneous data. Homogeneous refers to all elements in an array having the same data type. The main data structure for multidimensional arrays in NumPy is the `ndarray` class. In addition to the data stored in the array, this data structure also contains important metadata about the array, such as its shape, size, data type, and other attributes.

Attribute	Description
Shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
Size	The total number elements in the array.
Ndim	Number of dimensions (axes).
nbytes	Number of bytes used to store the data.
dtype	The data type of the elements in the array.

The following example demonstrates how these attributes are accessed for an instance data of the class `ndarray`:

```
>>> import numpy as np

# To create a NumPy array, you can use the function np.array()

>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> type(data)
numpy.ndarray
>>> data
array([[1, 2],
       [3, 4],
       [5, 6]])
```

	0	1
0	1	2
1	3	4
2	5	6

The following example demonstrates how these attributes are accessed for an instance data of the class `ndarray`:

```
>>> data.ndim
2
>>> data.shape
(3, 2)
>>> data.size
6
>>> data.dtype
dtype('int32')
>>> data.nbytes
24
```

	0	1
0	1	2
1	3	4
2	5	6

The following example demonstrates `np.zeros()`, `np.ones()`, `np.empty()`, `np.arange()`, `np.linspace()`


```
>>> np.zeros(2)
array([0., 0.])

>>> np.ones(2)
array([1., 1.])

>>> np.arange(4)
array([0, 1, 2, 3])
# specify the first number, last number, and the step size
>>> np.arange(2, 9, 2)
array([2, 4, 6, 8])
```

The following example demonstrates `np.zeros()`, `np.ones()`, `np.empty()`, `np.arange()`, `np.linspace()`

NumPy

```
>>> # Create an empty array with 2 elements
>>> np.empty(2)
    array([ 3.14, 42.   ])  # may vary
```

The function empty creates an array whose initial content is random and depends on the state of the memory. The reason to use empty over zeros (or something similar) is speed - just make sure to fill every element afterwards!

You can also use `np.linspace()` to create an array with values that are spaced linearly in a specified interval:

```
>>> np.linspace(0, 10, num=5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
# Specifying data type
>>> x = np.ones(2, dtype=np.int64)
>>> x
array([1, 1])
```

Adding, removing, and sorting elements: np.sort(), np.concatenate()

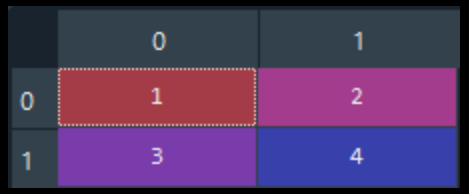

```
>>> arr = np.array([2, 1, 5, 3, 7, 4, 6, 8])
# sort the numbers in ascending order with:
>>> np.sort(arr)
array([1, 2, 3, 4, 5, 6, 7, 8])


>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([5, 6, 7, 8])
# concatenate them with np.concatenate()
>>> np.concatenate((a, b))
array([1, 2, 3, 4, 5, 6, 7, 8])
```

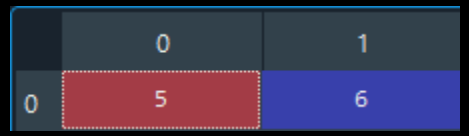
Adding, removing, and sorting elements: np.sort(), np.concatenate()

NumPy

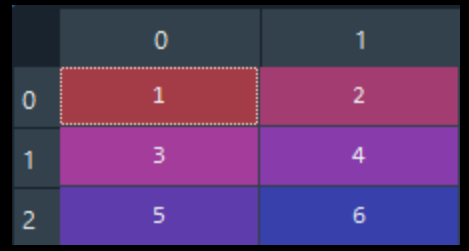
```
>>> x = np.array([[1, 2], [3, 4]])
>>> y = np.array([[5, 6]])
# concatenate them with:
>>> np.concatenate((x, y), axis=0)
array([[1, 2],
       [3, 4],
       [5, 6]])
```



	0	1
0	1	2
1	3	4



	0	1
0	5	6

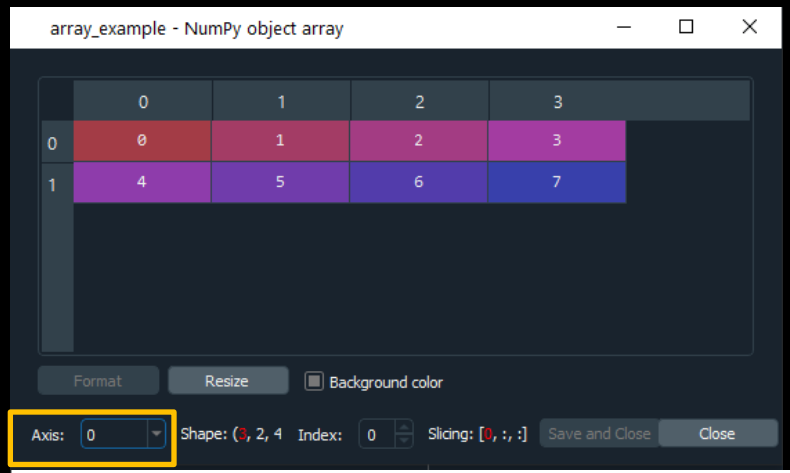


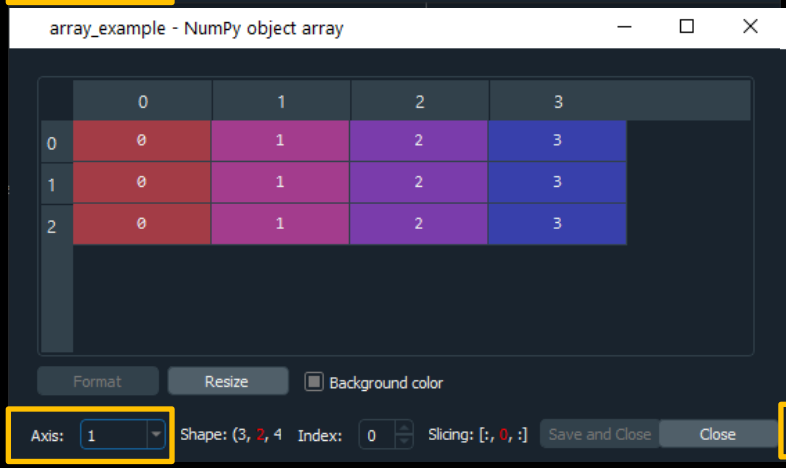
	0	1
0	1	2
1	3	4
2	5	6

```
>>> np.concatenate((x, y), axis=1)
ValueError: all the input array dimensions for the
  concatenation axis must match exactly, but along
  dimension 0, the array at index 0 has size 2 and the
  array at index 1 has size 1
```

How do you know the shape and size of an array?

```
>>> array_example = np.array([[[0, 1, 2, 3],
                               [4, 5, 6, 7]],
                              [[0, 1, 2, 3],
                               [4, 5, 6, 7]],
                              [[0 ,1 ,2, 3],
                               [4, 5, 6, 7]]])
```





How do you know the shape and size of an array?


```
# To find the number of dimensions of the array
>>> array_example.ndim
3
# To find the total number of elements in the array
>>> array_example.size
24
# And to find the shape of your array
>>> array_example.shape
(3, 2, 4)
```

How to reshape an array?


```
# start with an array with 6 element
>>> a = np.arange(6)
>>> print(a)
[0 1 2 3 4 5]
# make sure that new array also has a total of 6 elements
>>> b = a.reshape(3, 2)
>>> print(b)
[[0 1]
 [2 3]
 [4 5]]
```

How to convert a 1D array into a 2D array?

```
# start with an array with 6 element
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> a.shape
(6,)
# use np.newaxis to add a new axis
>>> a2 = a[np.newaxis, :]
>>> a2.shape
(1, 6)
>>> a3 = a[:, np.newaxis]
>>> a3.shape
(6, 1)
```

	0
0	1
1	2
2	3
3	4
4	5
5	6

column vector

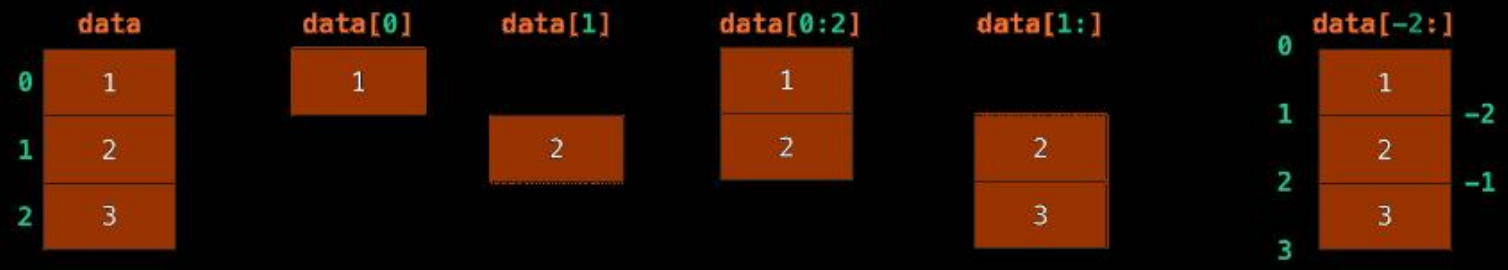
row vector

	0	1	2	3	4	5
0	1	2	3	4	5	6

Indexing and slicing

NumPy

```
# start with an array with 6 element
>>> data = np.array([1, 2, 3])
>>> data[1]
2
>>> data[0:2]
array([1, 2])
>>> data[1:]
array([2, 3])
>>> data[-2:]
array([2, 3])
```



Indexing and slicing


```
>>> a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
#print all of the values in the array that are less than 5
>>> print(a[a < 5])
[1 2 3 4]
>>> >>> five_up = (a >= 5)
>>> print(a[five_up])
[ 5  6  7  8  9 10 11 12]
>>> divisible_by_2 = a[a%2==0]
>>> print(divisible_by_2)
[ 2  4  6  8 10 12]
>>> c = a[(a > 2) & (a < 11)]
>>> print(c)
[ 3  4  5  6  7  8  9 10]
```

How to create an array from existing data?

NumPy

```
>>> a = np.array([1,  2,  3,  4,  5,  6,  7,  8,  9, 10])
>>> arr1 = a[3:8]
>>> arr1
array([4, 5, 6, 7, 8])
>>> a1 = np.array([[1, 1],
...                [2, 2]])
>>> a2 = np.array([[3, 3],
...                [4, 4]])
>>> np.vstack((a1, a2))
array([[1, 1],
       [2, 2],
       [3, 3],
       [4, 4]])
```

How to create an array from existing data?


```
>>> np.hstack((a1, a2))
array([[1, 1, 3, 3],
       [2, 2, 4, 4]])
>>> x = np.arange(1, 25).reshape(2, 12)
>>> x
array([[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12],
       [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]])
```

Basic array operations

NumPy

`data = np.array([1,2])`

data

1
2

`ones = np.ones(2)`

ones

1
1

```
>>> data = np.array([1, 2])
>>> ones = np.ones(2, dtype=int)
>>> data + ones
array([2, 3])
```

`data + ones` =

data		ones		
1	+	1	=	2
2		1		3

Basic array operations

```
>>> data - ones
array([0, 1])
```


```
>>> data * data
array([1, 4])
```

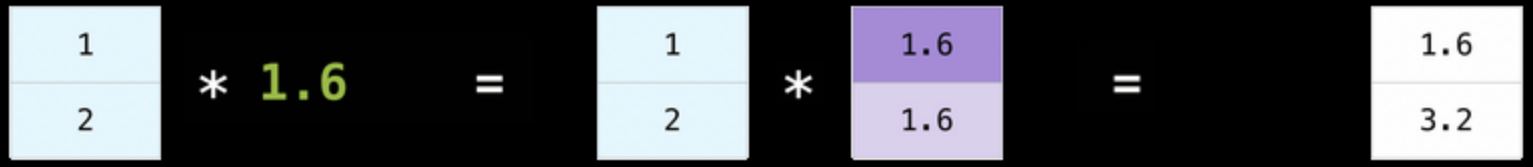

```
>>> data / data
array([1., 1.])
```


```
>>> a = np.array([1, 2, 3, 4])
>>> a.sum()
10
>>> b = np.array([[1, 1], [2, 2]])
>>> b.sum(axis=0)
array([3, 3])
>>> b.sum(axis=1)
array([2, 4])
```

an operation between a vector and a scalar

NumPy

```
>>> data = np.array([1.0, 2.0])
>>> data * 1.6
array([1.6, 3.2])
```



NumPy understands that the multiplication should happen with each cell. That concept is called **broadcasting**. Broadcasting is a mechanism that allows NumPy to perform operations on arrays of different shapes.

Accessing elements of matrices

NumPy

```
>>> data = np.array([[1, 2], [3, 4], [5, 6]])
>>> data
array([[1, 2],
       [3, 4],
       [5, 6])
```

data

	0	1
0	1	2
1	3	4
2	5	6

data[0,1]

	0	1
0	1	2
1	3	4
2	5	6

data[1:3]

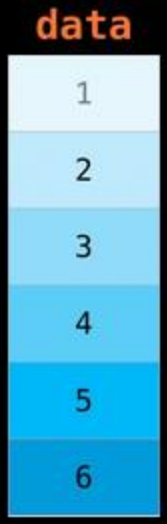
	0	1
0	1	2
1	3	4
2	5	6

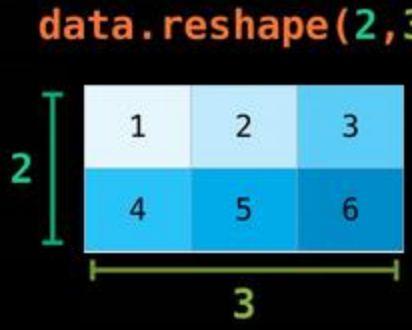
data[0:2,0]

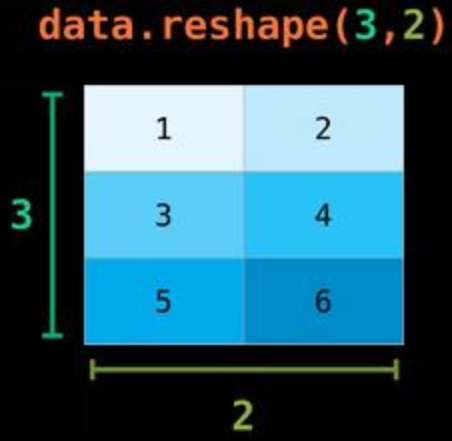
	0	1
0	1	2
1	3	4
2	5	6

Transposing and reshaping a matrix

NumPy








```
>>> data.reshape(2, 3)
array([[1, 2, 3],
       [4, 5, 6]])
>>> data.reshape(3, 2)
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Transposing and reshaping a matrix

NumPy

data

1	2
3	4
5	6

data.T

1	3	5
2	4	6

```
>>> data = np.arange(1,7).reshape(3,2)
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> data.T # data.transpose() can also be used
array([[1, 3, 5],
       [2, 4, 6]])
```

How to reverse an array

NumPy

```
# np.flip() function allows you to flip, or reverse
>>> data = np.array([1, 2, 3, 4, 5, 6, 7, 8])
>>> reversed_data = np.flip(data)
>>> print('Reversed Array: ', reversed_data)
Reversed Array:  [8 7 6 5 4 3 2 1]
>>> data_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8],
                                      [9, 10, 11, 12]])
>>> reversed_data2d_rows = np.flip(data_2d, axis=0)
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

	0	1	2	3
0	9	10	11	12
1	5	6	7	8
2	1	2	3	4

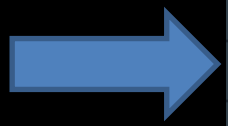
How to reverse an array

NumPy

```
>>> reversed_data2d_columns = np.flip(data_2d, axis=1)
```

	0	1	2	3
0	4	3	2	1
1	8	7	6	5
2	12	11	10	9

```
>>> # reverse only the rows
>>> data_2d[1] = np.flip(data_2d[1])
```



	0	1	2	3
0	1	2	3	4
1	8	7	6	5
2	9	10	11	12

Unique items and counts

NumPy

```
>>> a = np.array([11, 11, 12, 13, 14, 15, 16, 17,
                  12, 13, 11, 14, 18, 19, 20])
>>> unique_values = np.unique(a)
>>> print(unique_values)
[11 12 13 14 15 16 17 18 19 20]
>>> >>> unique_values, indices_list = np.unique(a,
    return_index = True)
>>> print(indices_list)
[ 0  2  3  4  5  6  7 12 13 14]
>>> >>> unique_values, occurrence_count = np.unique(a,
    return_counts = True)
>>> print(occurrence_count)
[3 2 2 2 1 1 1 1 1 1]
```

Display data as an image

`matplotlib.pyplot.imshow`

Matplotlib: Visualization with Python

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Matplotlib makes easy things easy and hard things possible.

```
>>> import matplotlib.pyplot as plt
```

https://matplotlib.org/stable/index.html

Display data as an image

`matplotlib.pyplot.imshow`

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image set up the color mapping using the parameters cmap='gray', vmin=0, vmax=255.

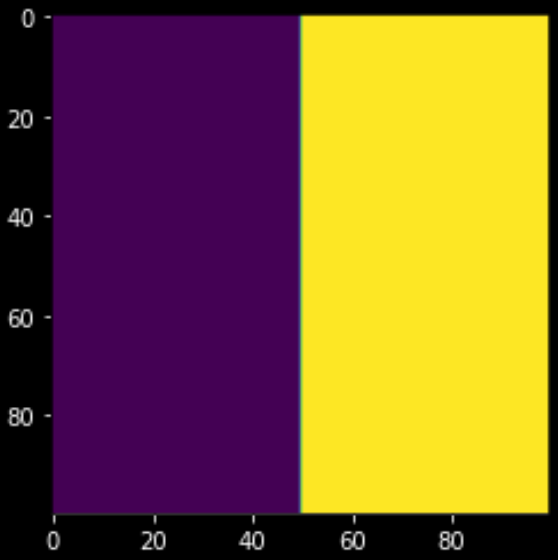
```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> beyaz = np.ones((100,100))
>>> beyaz[:, 50:100] = 0
>>> fig = plt.figure()
>>> plt.imshow(beyaz)
>>> plt.show()
```

Display data as an image

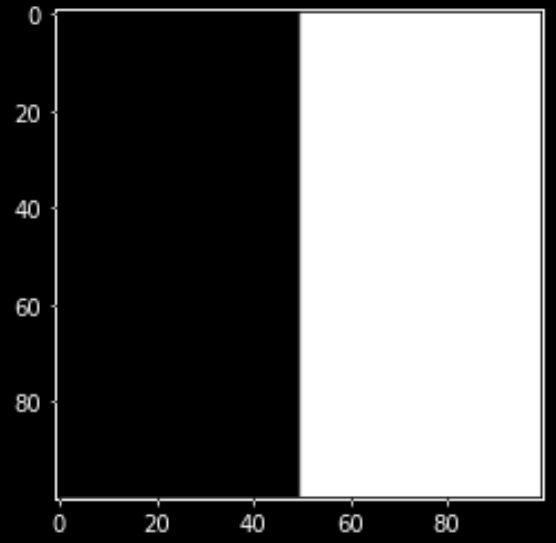
`matplotlib.pyplot.imshow`

NumPy

The input may either be actual RGB(A) data, or 2D scalar data, which will be rendered as a pseudocolor image. For displaying a grayscale image set up the color mapping using the parameters cmap='gray', vmin=0, vmax=255.



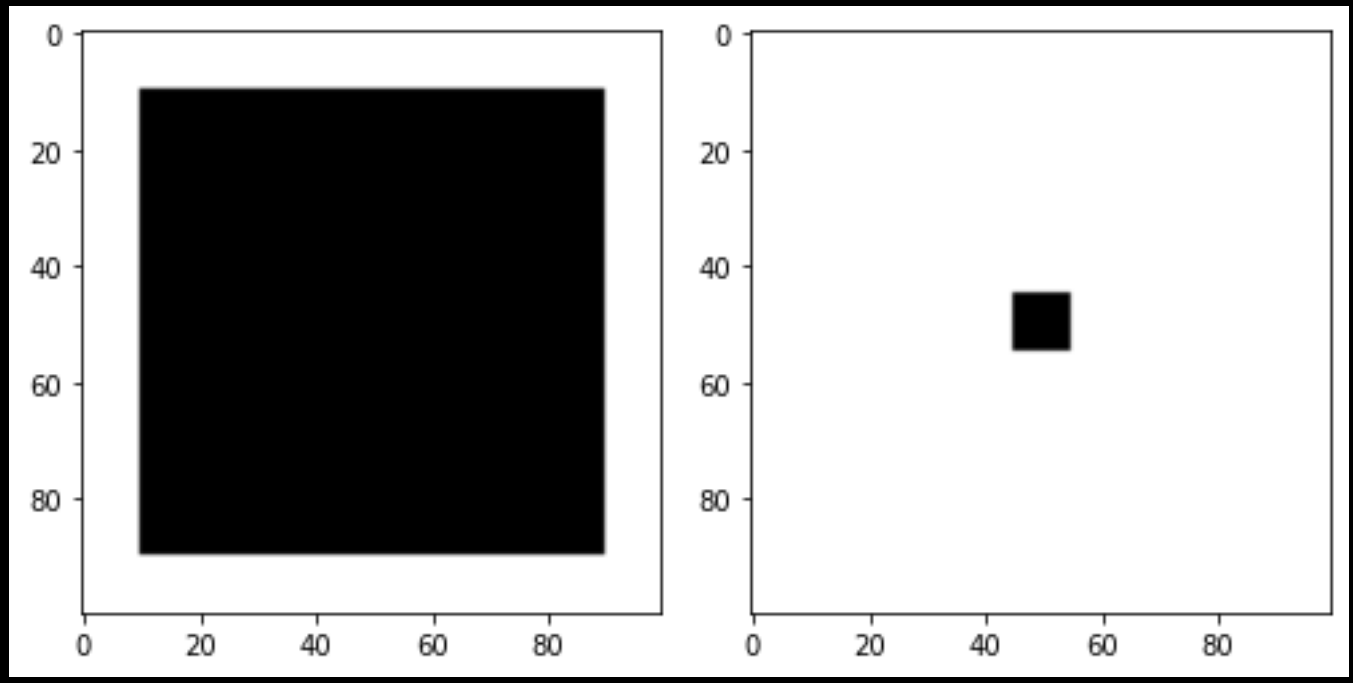
rendered as a pseudocolor




```
plt.imshow(beyaz, cmap='gray', vmin=0, vmax=1)
```

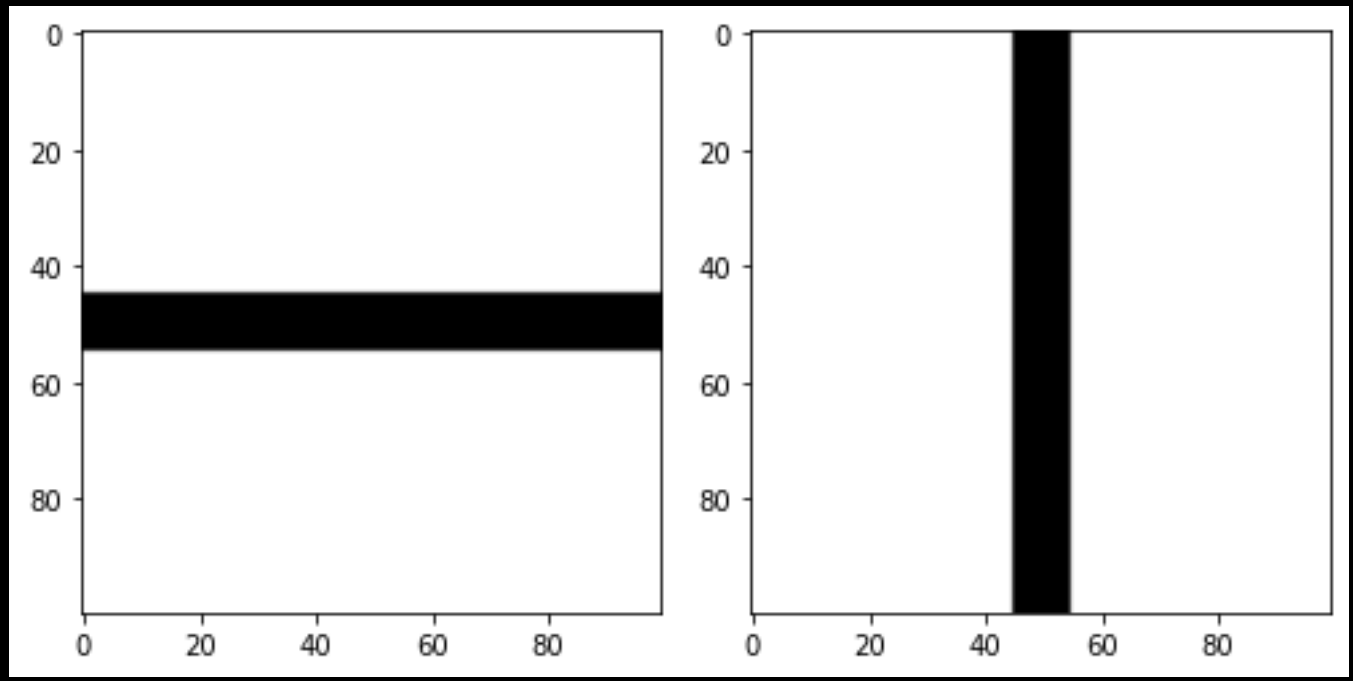
Homework

Create two-dimensional arrays and plot them as in given figures.



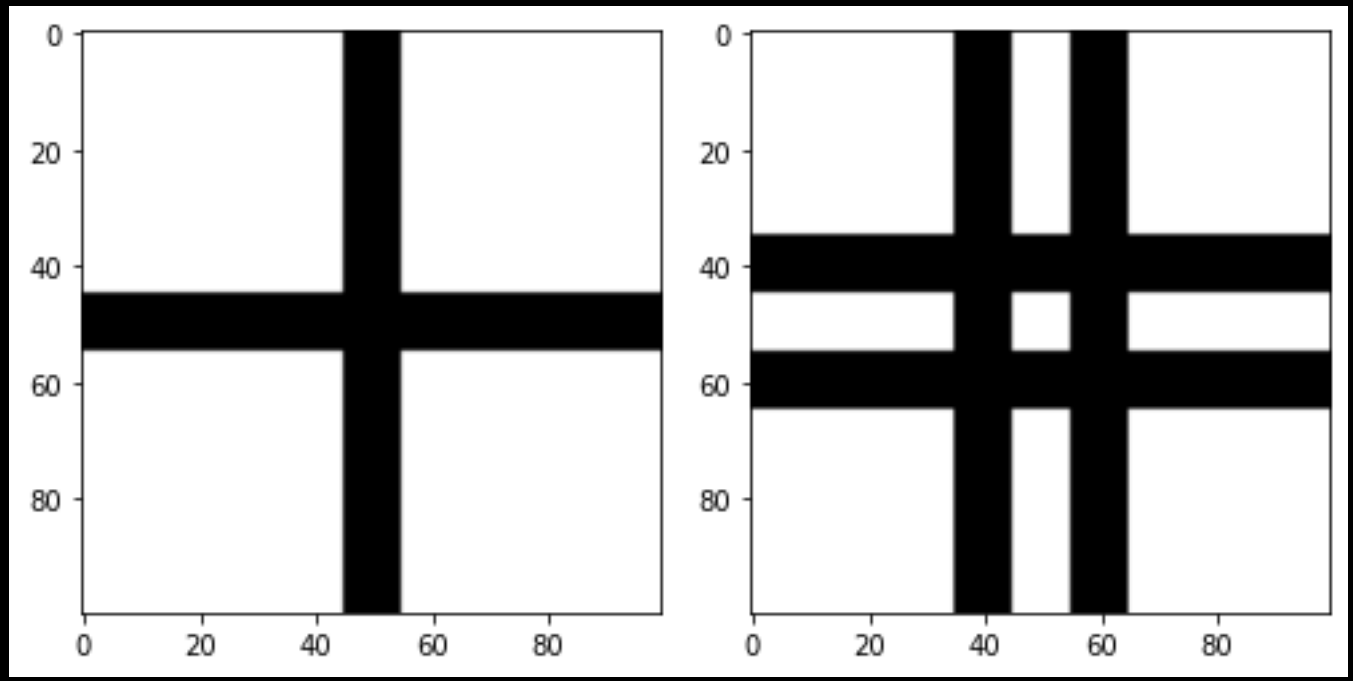
Homework

Create two-dimensional arrays and plot them as in given figures.



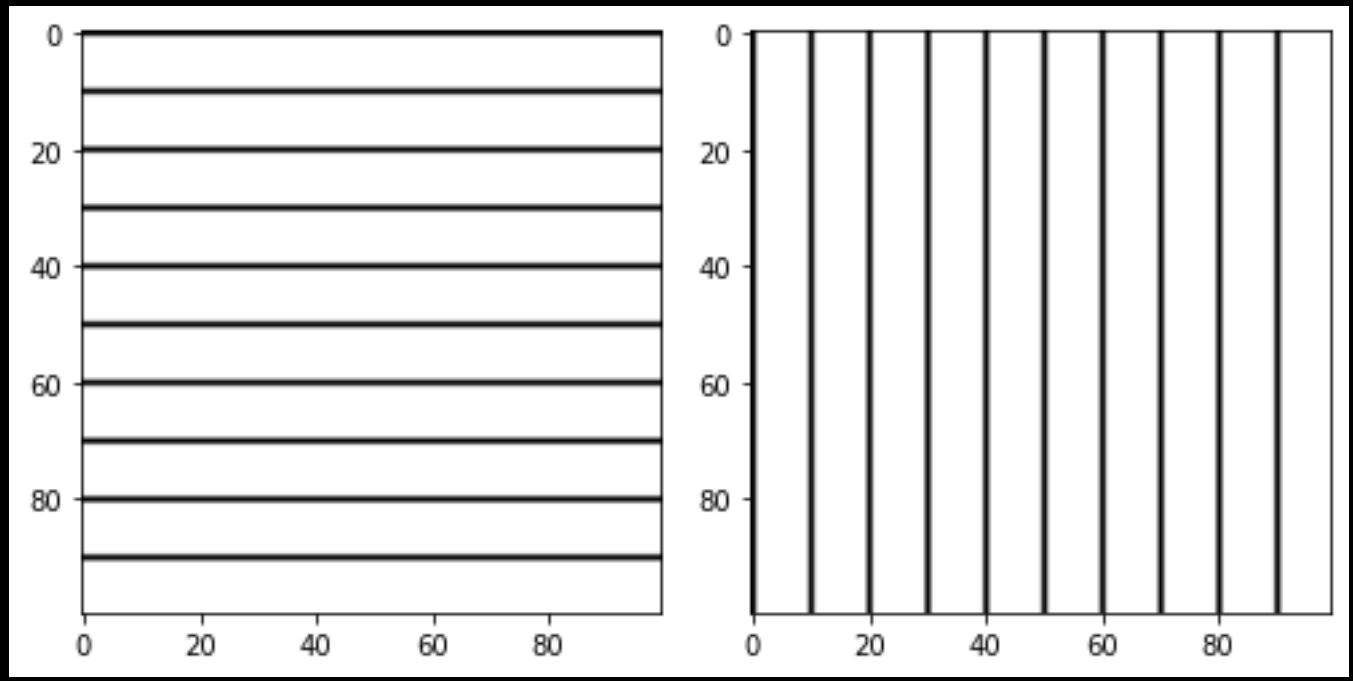
Homework

Create two-dimensional arrays and plot them as in given figures.



Homework

Create two-dimensional arrays and plot them as in given figures.



Homework

Create two-dimensional arrays and plot them as in given figures.