Intro to Python

Serafina Di Gioia
Postdoctoral researcher in ML @ICTP

Plan for this lecture

15 min

Intro to Python

- identikit
- comparison with compiled language
- built-in types
- classes
- decorators

10 min

Numpy

- scope of the library
- core objects
- examples

10 min

Scipy

- objectives
- syntax/functions
- examples.

10 min

Cupy

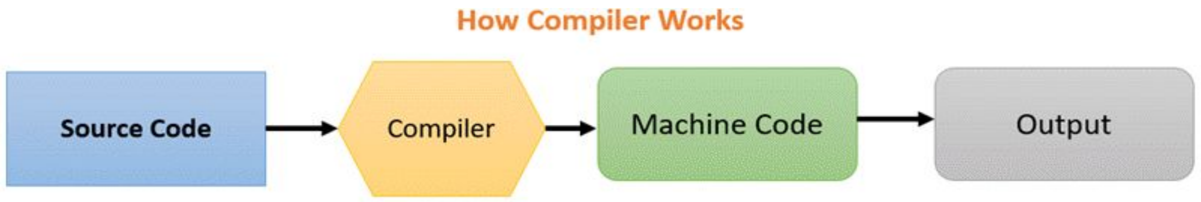
Lab activity

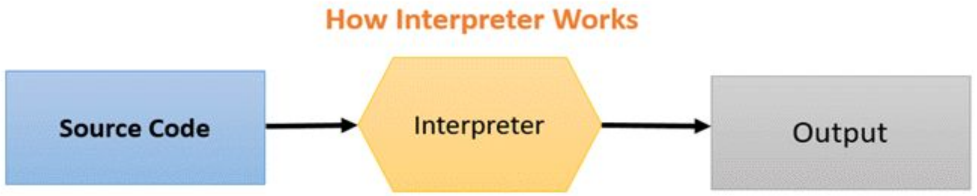
Python’s identikit

Python is…

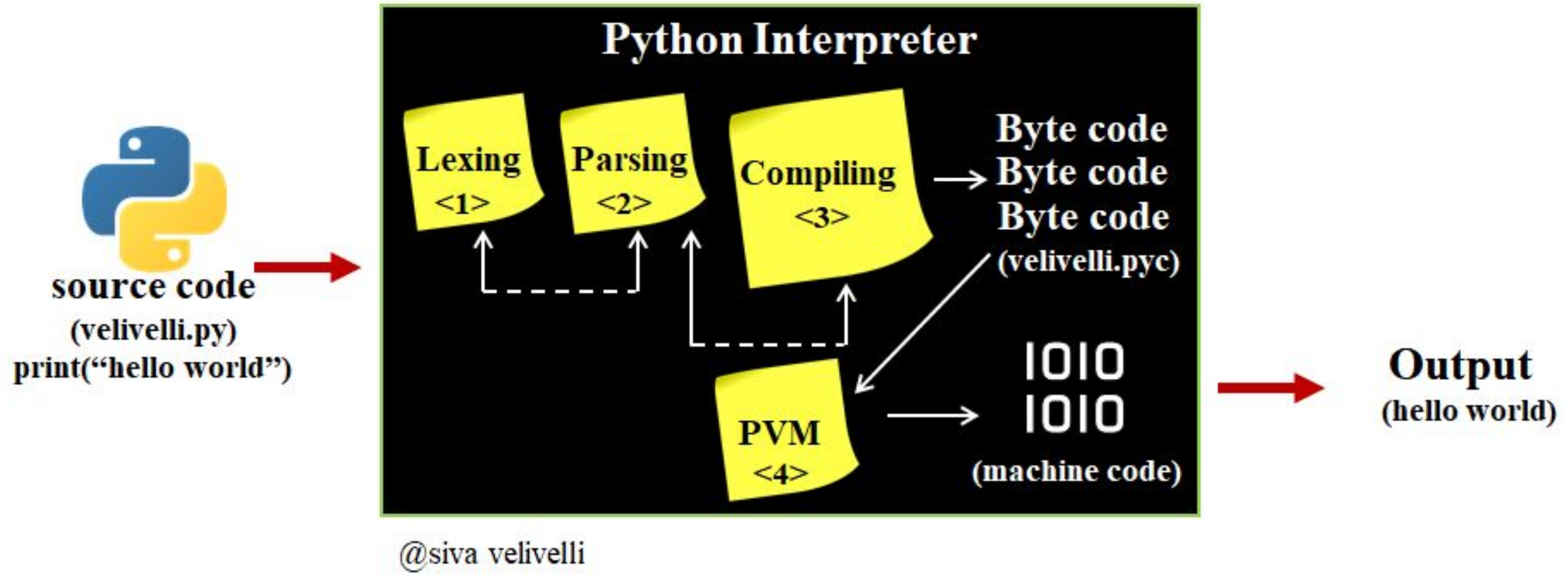
- named after the BBC show “Monty Python’s Flying Circus”
- high-level interpreted language
- object-oriented programming language
- easy of use
- extensible (since it is based on modules)

Why interpreted codes like Python are slower?

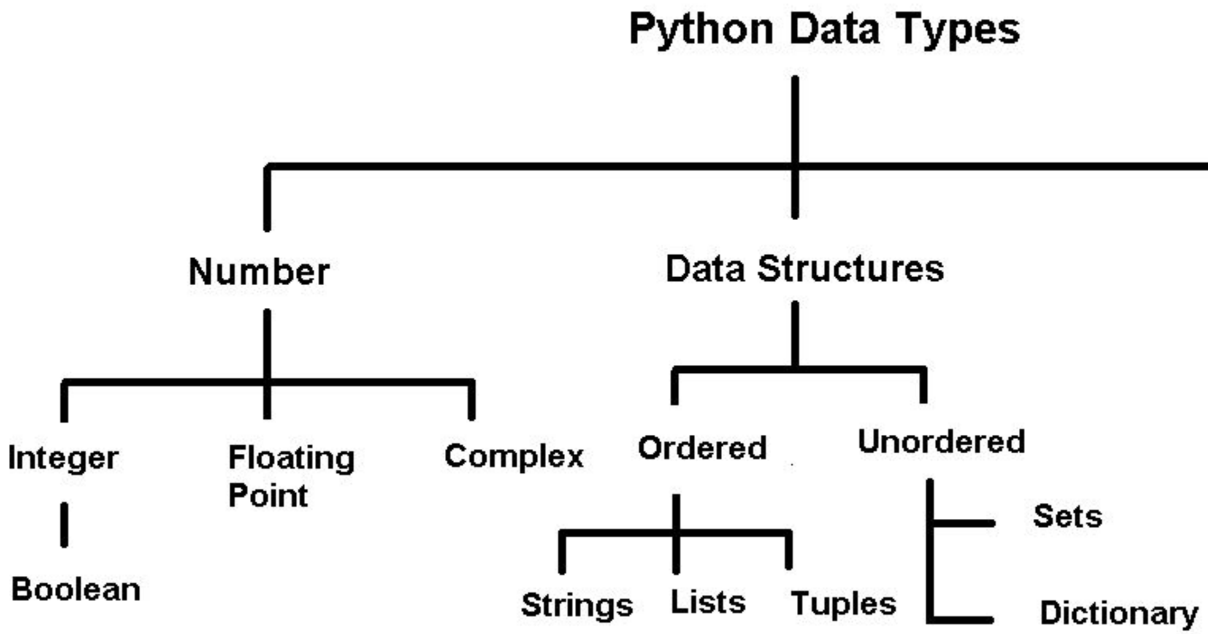




The Python interpreter



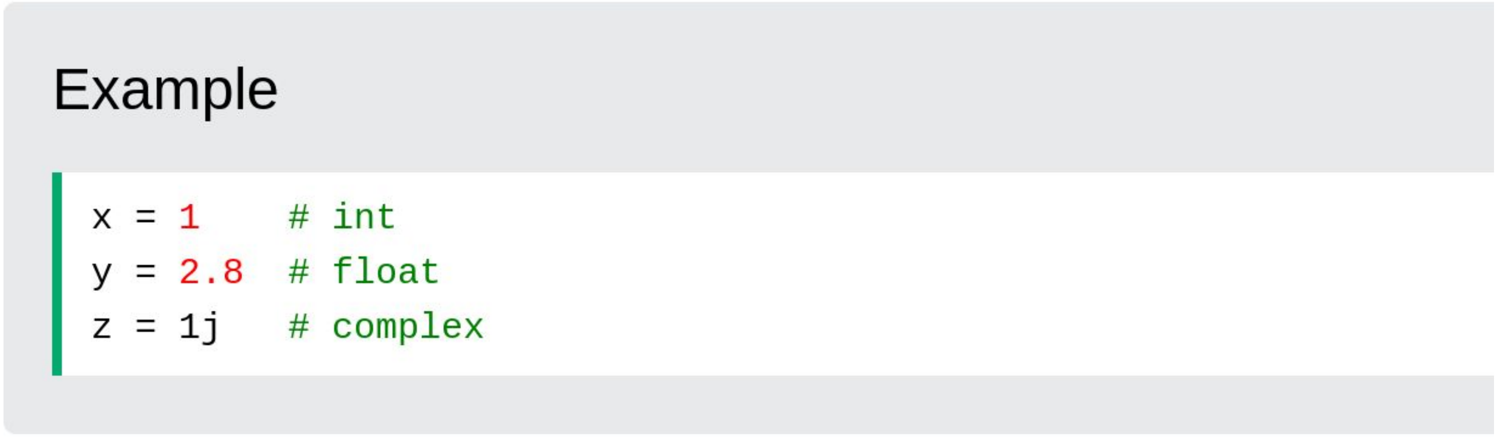
Python data types



Number types in Python

- `int`
- `float`
- `complex`

Variables of numeric types are created when you assign a value to them:



Example

```
x = 1    # int
y = 2.8  # float
z = 1j   # complex
```

What is a string in Python?

A string may not span across multiple lines or contain a " character.

```
"This is not
a legal String."
```

```
"This is not a "legal" String either."
```

A string can represent characters by preceding them with a backslash.

- `\t` tab character
- `\n` new line character
- `\"` quotation mark character
- `\\` backslash character

- Example: `"Hello\tthere\nHow are you?"`

How do we access chars in string in Python?

Characters in a string are numbered with *indexes* starting at 0:

- Example:

```
name = "P. Diddy"
```

index	0	1	2	3	4	5	6	7
character	P	.		D	i	d	d	y

Accessing an individual character of a string:

variableName [***index***]

- Example:

```
print name, "starts with", name[0]
```

Output:

```
P. Diddy starts with P
```

What is a list in Python?

- ordered collection of data
- can contain different types of elements
- mutable object
- share all the methods of strings plus the methods in next slide

Lists methods

- .append(element)
- .insert(i, element)
- .remove(element) -> removes 1st element with this val
- .pop(i)
- .index(element)
- .count()

What is a list in Python?

- ordered collection of data
- can contain different types of elements
- mutable object
- share all the methods of strings plus the methods in next slide
- can be constructed in 2

  ways

1)
```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```

2)
```
thislist = list(("apple", "banana", "cherry"))
print(thislist)
```

What is a tuple in Python?

- ordered collection of data
- can contain different types of elements
- unmutable object
- can be constructed in 2 ways
- only two built-in methods:

  count() and index()

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

```
thistuple = tuple(("apple", "banana", "cherry"))
print(thistuple)
```

What are dictionaries in Python?

- ordered data structured since Python 3.7
- changeable
- to access the objects in the dictionary you need to use a key
- do not accept duplicate associated to the same key

```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

What are sets in Python?

- collection of item
- unordered, unchangeable*, and unindexed
- No duplicate members

What are functions in Python?

Functions in Python are first class objects

Quoting the creator of Python, Guido Van Rossum: "One of my goals for Python was to make it so that all objects were "first class." By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, etc.) to have equal status."

(from "The History of Python," February 27, 2009.)

Objects in Python are classified by their value, type, and identity (aka. memory address).

Being a first class object means:

1. it can be created at runtime.
2. it can be assigned to a variable.
3. it can be passed as a argument to a function.
4. it can be returned as a result from a function.
5. it can have properties and methods

How do we define a function?

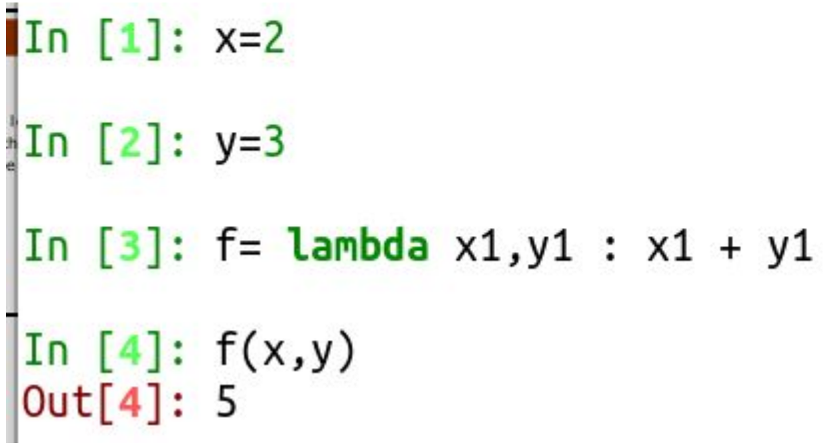
```python
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

From a function's perspective:

A parameter is the variable listed inside the parentheses in the function definition.

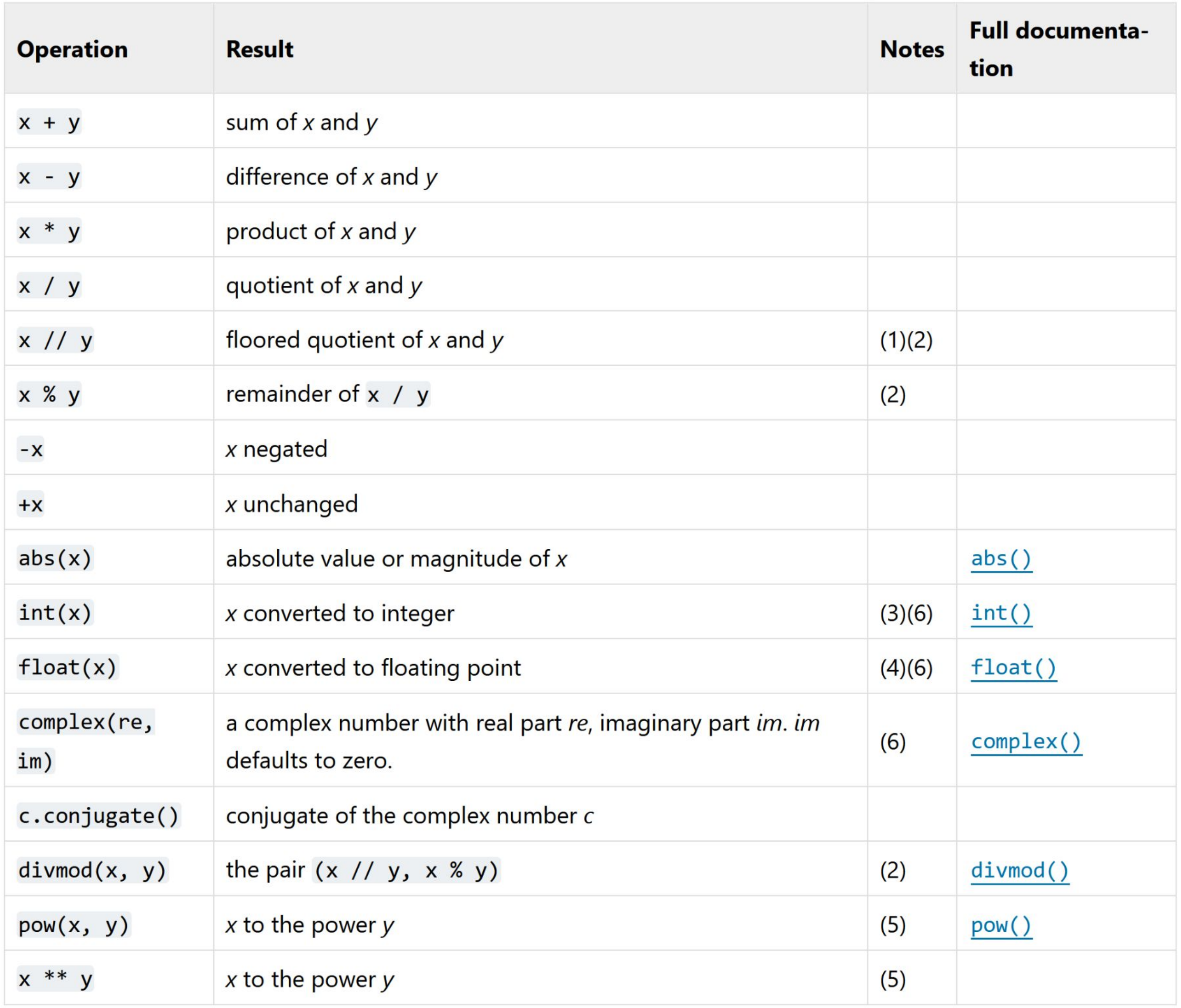
An argument is the value that is sent to the function when it is called.

Lambda functions:



```
In [1]: x=2

In [2]: y=3

In [3]: f= lambda x1,y1 : x1 + y1

In [4]: f(x,y)
Out[4]: 5
```

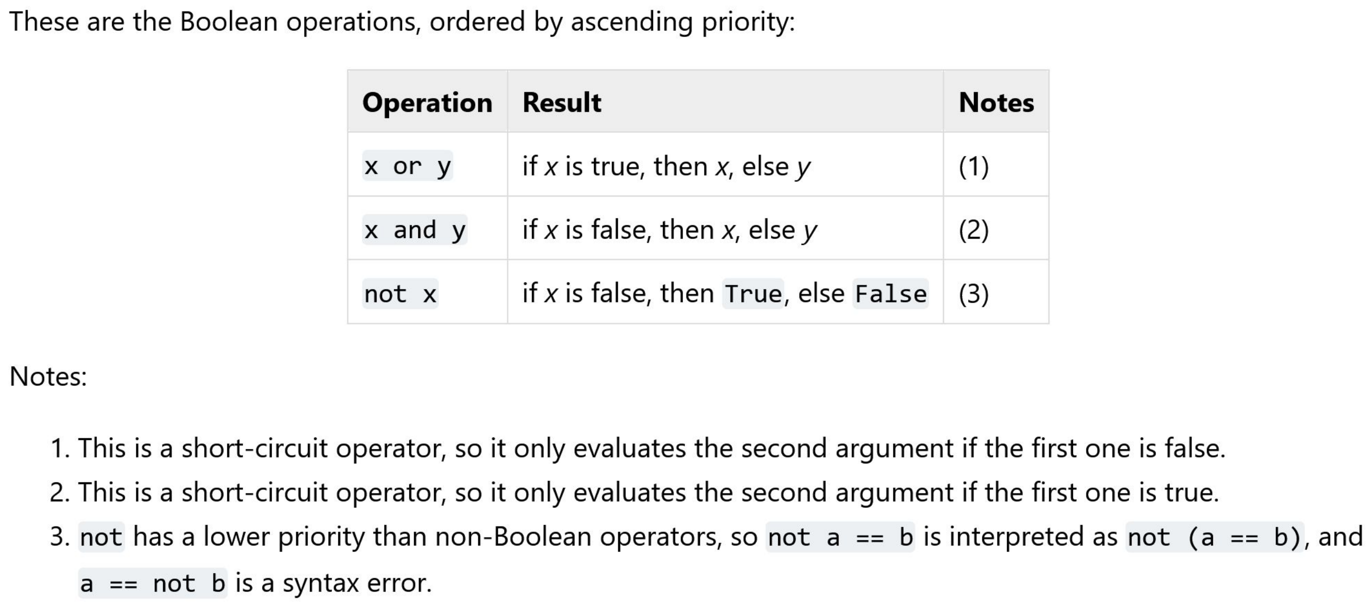
Python arithmetics



Operation	Result	Notes	Full documentation
`x + y`	sum of *x* and *y*		
`x - y`	difference of *x* and *y*		
`x * y`	product of *x* and *y*		
`x / y`	quotient of *x* and *y*		
`x // y`	floored quotient of *x* and *y*	(1)(2)	
`x % y`	remainder of `x / y`	(2)	
`-x`	*x* negated		
`+x`	*x* unchanged		
`abs(x)`	absolute value or magnitude of *x*		`abs()`
`int(x)`	*x* converted to integer	(3)(6)	`int()`
`float(x)`	*x* converted to floating point	(4)(6)	`float()`
`complex(re, im)`	a complex number with real part *re*, imaginary part *im*. *im* defaults to zero.	(6)	`complex()`
`c.conjugate()`	conjugate of the complex number *c*		
`divmod(x, y)`	the pair `(x // y, x % y)`	(2)	`divmod()`
`pow(x, y)`	*x* to the power *y*	(5)	`pow()`
`x ** y`	*x* to the power *y*	(5)	

Python comparison operations

for boolean objects…



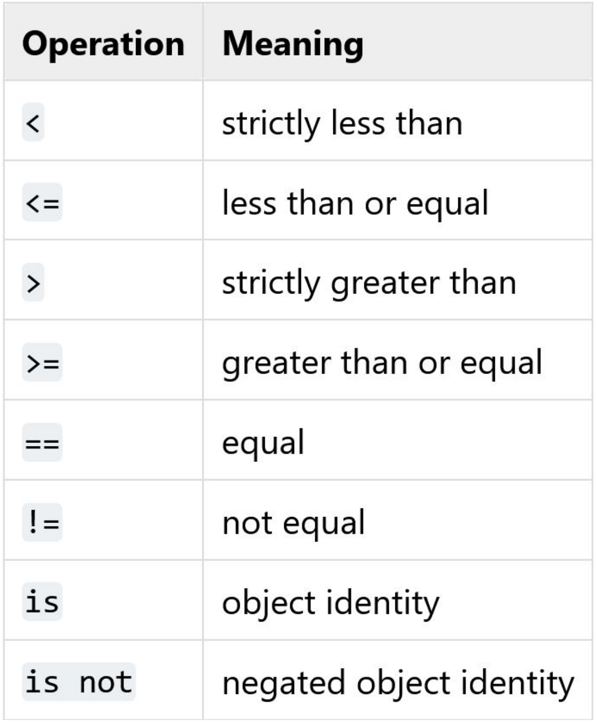
These are the Boolean operations, ordered by ascending priority:

Operation	**Result**	**Notes**
`x or y`	if *x* is true, then *x*, else *y*	(1)
`x and y`	if *x* is false, then *x*, else *y*	(2)
`not x`	if *x* is false, then `True`, else `False`	(3)

Notes:

1. This is a short-circuit operator, so it only evaluates the second argument if the first one is false.
2. This is a short-circuit operator, so it only evaluates the second argument if the first one is true.
3. `not` has a lower priority than non-Boolean operators, so `not a == b` is interpreted as `not (a == b)`, and `a == not b` is a syntax error.

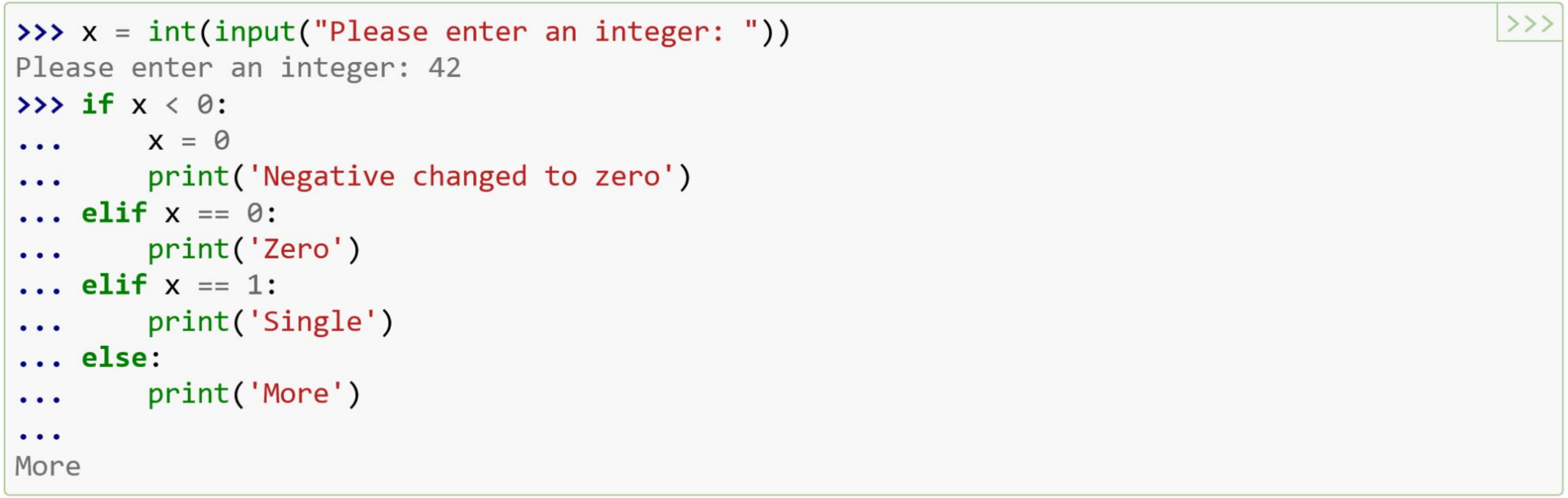
Python comparison operations for numeric type objects



Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal
`!=`	not equal
`is`	object identity
`is not`	negated object identity

Python Control-Flow statements

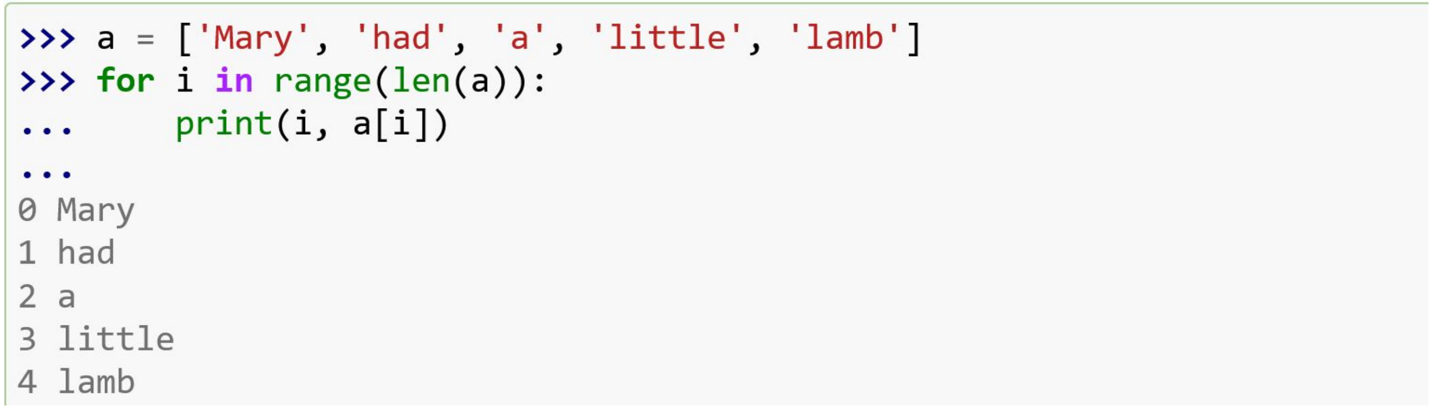
if-else statements



```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

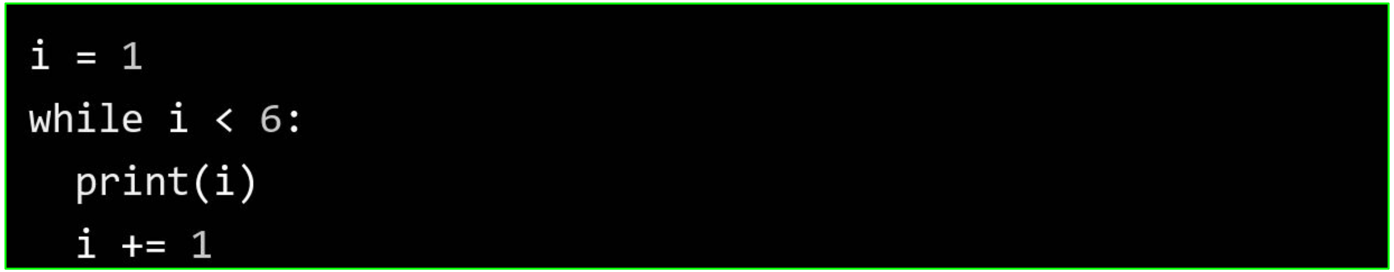
Python Control-Flow statements

for statement:



```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

while statement:



```
i = 1
while i < 6:
  print(i)
  i += 1
```

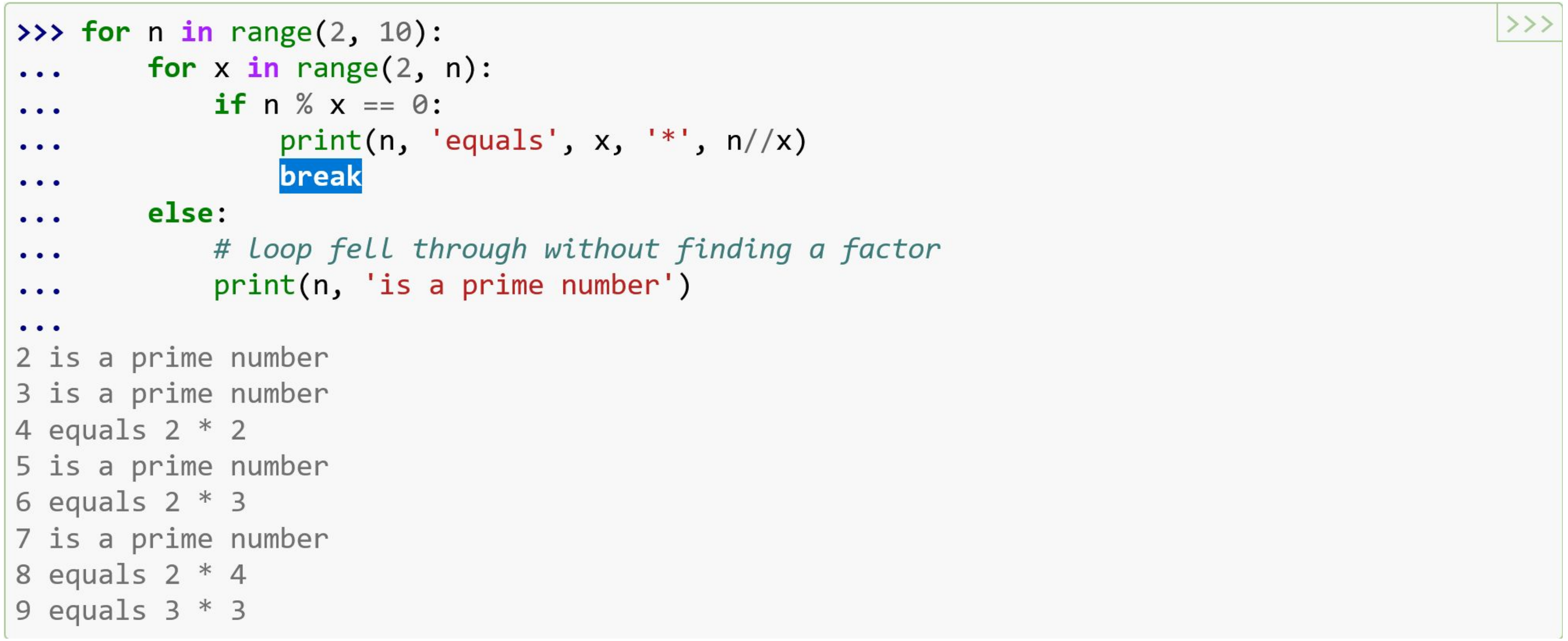
Python Control-Flow statements

single statement conditions:

break	Jumps out of the closest enclosing loop
continue	Jumps to the top of the closest enclosing loop
pass	Does nothing, empty statement placeholder

Python Control-Flow statements

indented loops and break conditions:



```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Classes in Python…

- Classes provide a means of creating user-defined types and functionalities,allowing to bundle objects and functionalities together.
- Creating a new class creates a new *type* of object, allowing new *instances* of that type to be made.
- Class can be defined uscing the Class() object and __init__ constructors
- Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

Typical constructs:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

- In practice, the statements inside a class definition will usually be function definition
- Class objects support two kinds of operations: attribute references and instantiation.
  - *Attribute references* use the standard syntax used for all attribute references in Python: `obj.name`.
  - Instantiation uses function notation : Class()

How to define a class in Python

```
from math import *

class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return sqrt(self.x * self.x + self.y * self.y)

    def distance(self, other):
        dx = self.x - other.x
        dy = self.y - other.y
        return sqrt(dx * dx + dy * dy)

    def move(self, dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return "(" + str(self.x) + ", " + str(self.y) + ")"
```

Class variables VS instance variables

```
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'
```

What happens when you create an instance of a Python class?

- A block of memory is allocated on the heap. The size of memory allocated is decided by the attributes and methods available in that class(Moto).
- After the memory block is allocated, the special method __init__() is called internally. Initial data is stored in the variables through this method.
- The location of the allocated memory address of the instance is returned to the object(Moto).
- The memory location is passed to self.

Class inheritance

A children class inherits the methods of the parent class but can override them

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

Example of match statement used on user-defined class

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Origin")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Somewhere else")
        case _:
            print("Not a point")
```

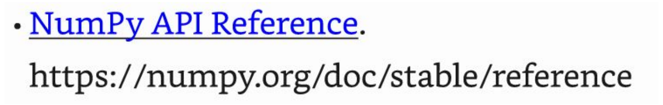
How to access Python interactive docstring

In ipython you can simply type a ? after an object or a function

Example:

Numpy

- library dedicated to scientific computing
- open-source
- available since
- core object: ndarray (numeric array)



• NumPy API Reference.

https://numpy.org/doc/stable/reference

Why numpy matters?

1. Efficiency
2. Ease of use
3. vectorization

What is a numpy array?

An array is a grid of values. It contains information about the raw data, how to locate an element, and how to interpret an element.

It can be indexed by

- a tuple of nonnegative integers
- booleans
- another array
- by integers.

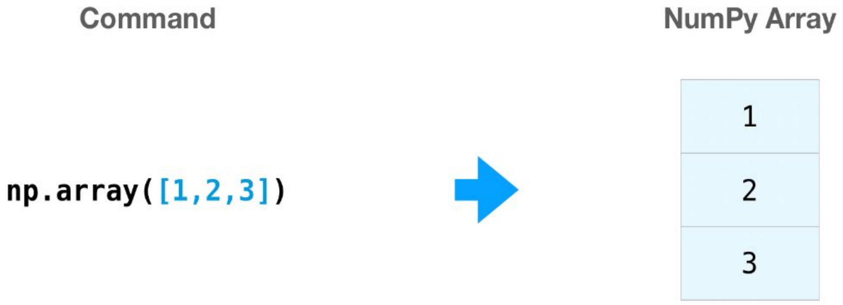
It has the following attributes:

1. `ndim, it` will tell you the number of axes, or dimensions, of the array.
2. `shape` of the array. It is a tuple of integers giving the size of the array along each dimension.
3. size, product of the elements in array shape
4. dtype, type of elements in the array

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
```

to initialize a numpy.array() we use a list, or a tuple of lists

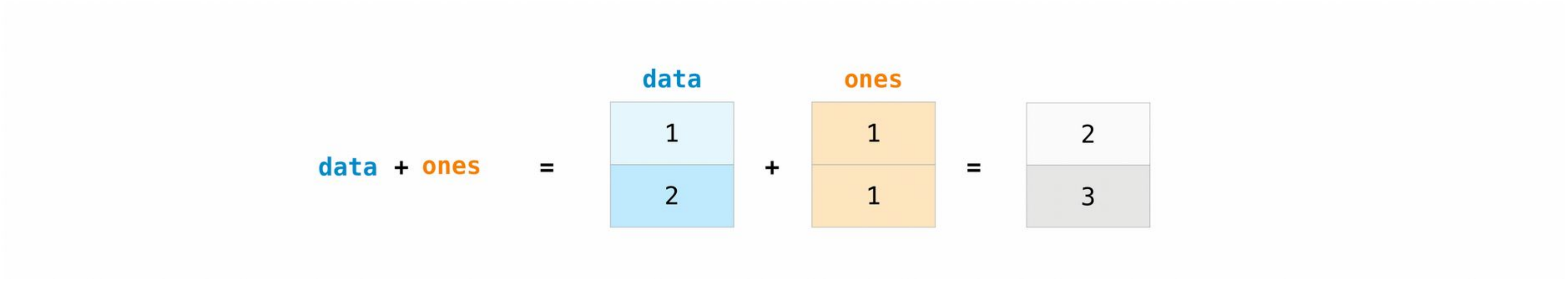
Visualizing a Numpy array



```
np.array([1,2,3])
```

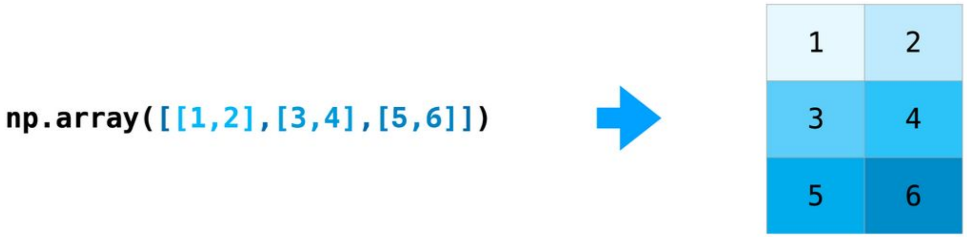
Vector addition in numpy

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



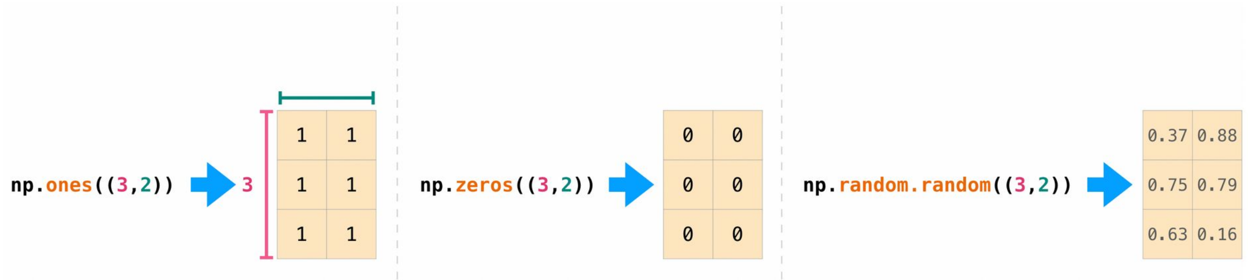
Creating a 2D array (also called a matrix)

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



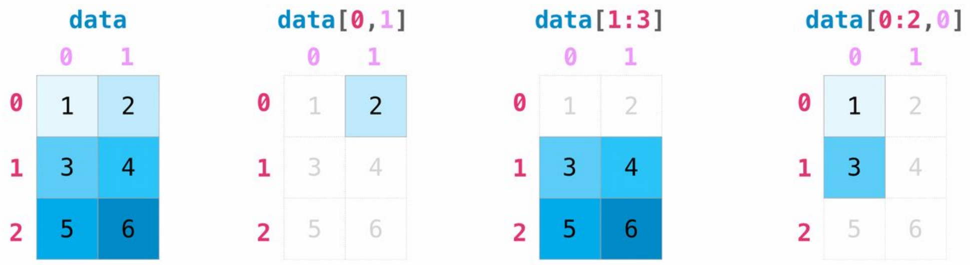
Filling a matrix

```
>>> np.ones((3, 2))
array([[1., 1.],
       [1., 1.],
       [1., 1.]])
>>> np.zeros((3, 2))
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
>>> rng.random((3, 2))
array([[0.01652764, 0.81327024],
       [0.91275558, 0.60663578],
       [0.72949656, 0.54362499]])  # may vary
```



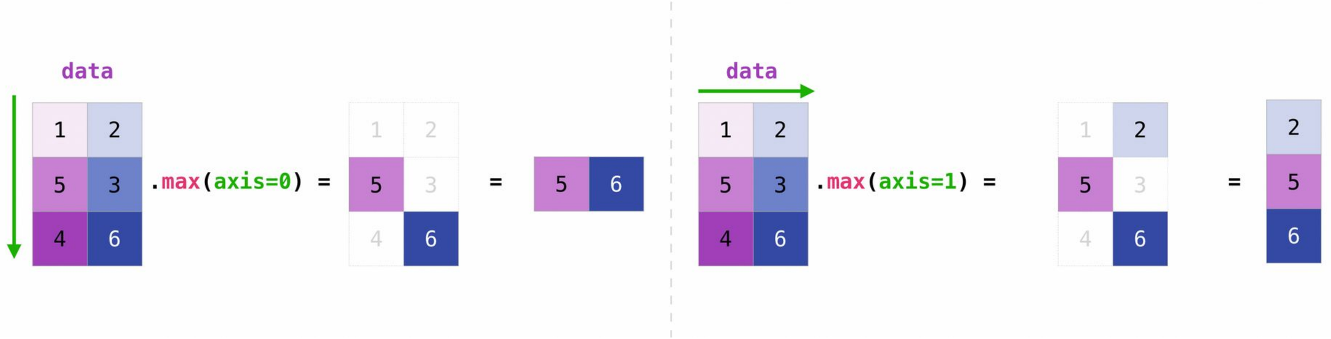
Slicing a matrix

```
>>> data[0, 1]
2
>>> data[1:3]
array([[3, 4],
       [5, 6]])
>>> data[0:2, 0]
array([1, 3])
```



Operations on a single axis of a matrix

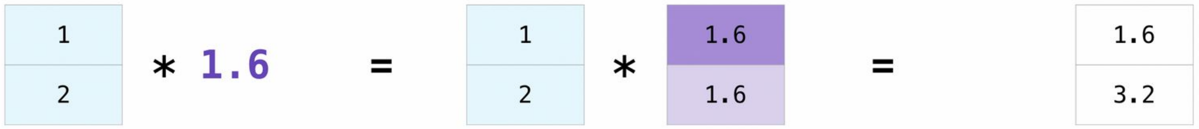
```
>>> data = np.array([[1, 2], [5, 3], [4, 6]])
>>> data
array([[1, 2],
       [5, 3],
       [4, 6]])
>>> data.max(axis=0)
array([5, 6])
>>> data.max(axis=1)
array([2, 5, 6])
```



Broadcasting

There are times when you might want to carry out an operation between an array and a single number (also called *an operation between a vector and a scalar*) or between arrays of two different sizes. For example, your array (we'll call it "data") might contain information about distance in miles but you want to convert the information to kilometers. You can perform this operation with:

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



I/O with Numpy

```
>>> csv_arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
```

You can easily save it as a .csv file with the name “new_file.csv” like this:

```
>>> np.savetxt('new_file.csv', csv_arr)
```

You can quickly and easily load your saved text file using `loadtxt()`:

```
>>> np.loadtxt('new_file.csv')
array([1., 2., 3., 4., 5., 6., 7., 8.])
```

The `savetxt()` and `loadtxt()` functions accept additional optional parameters such as header, footer, and delimiter. While text files can be easier for sharing, .npy and .npz files are smaller and faster to read. If you need more sophisticated handling of your text file (for example, if you need to work with lines that contain missing values), you will want to use the **`genfromtxt`** function.

You can save it as “filename.npy” with:

```
>>> np.save('filename', a)
```

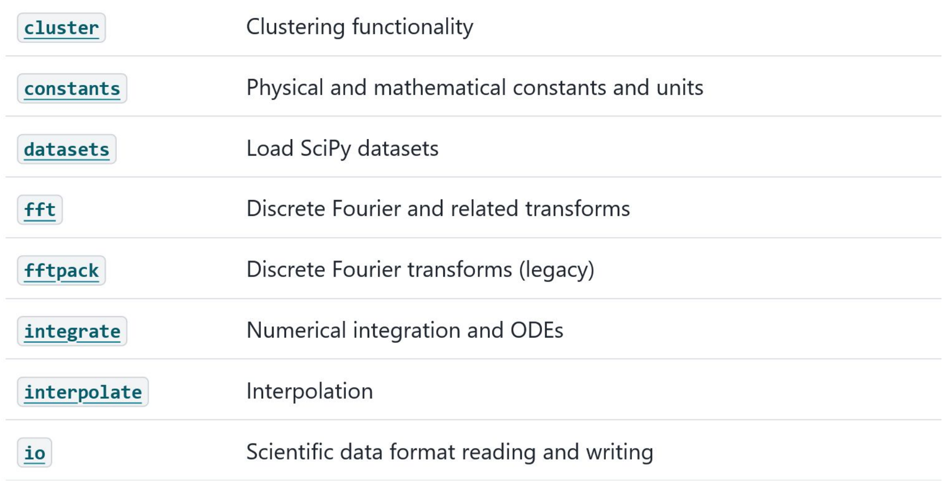
You can use `np.load()` to reconstruct your array.

```
>>> b = np.load('filename.npy')
```

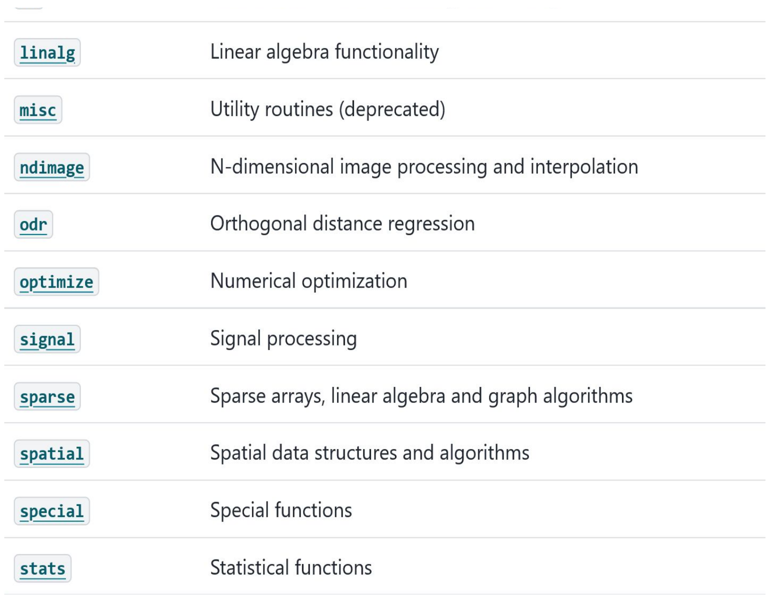
Scipy

SciPy is a collection of mathematical algorithms and convenience functions built on NumPy . It adds significant power to Python by providing the user with high-level commands and classes for manipulating and visualizing data.

Scipy submodules



`cluster`	Clustering functionality
`constants`	Physical and mathematical constants and units
`datasets`	Load SciPy datasets
`fft`	Discrete Fourier and related transforms
`fftpack`	Discrete Fourier transforms (legacy)
`integrate`	Numerical integration and ODEs
`interpolate`	Interpolation
`io`	Scientific data format reading and writing

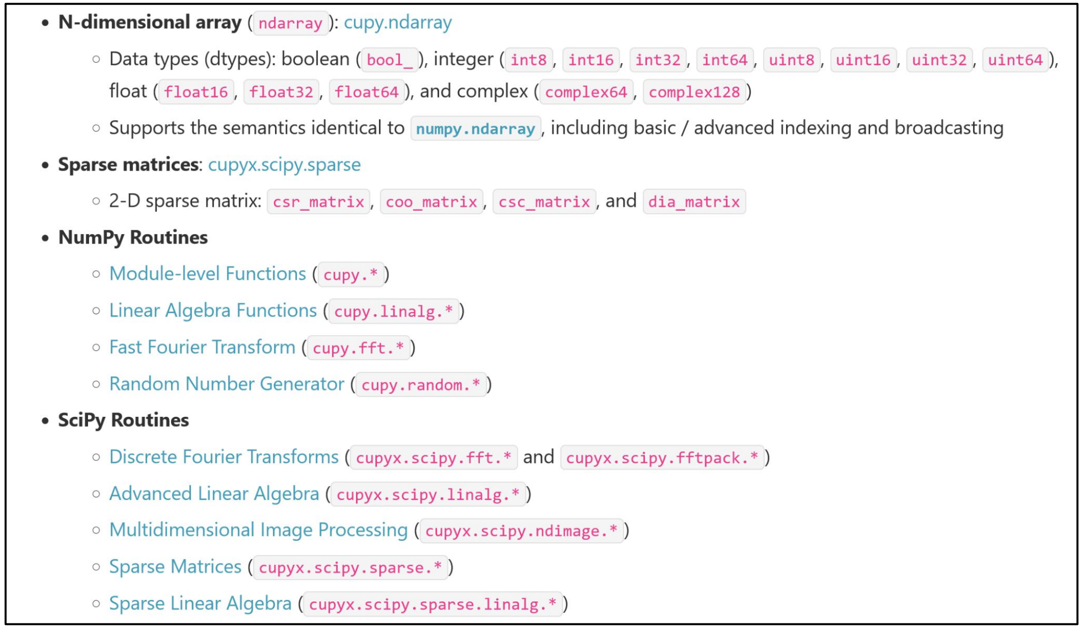


`linalg`	Linear algebra functionality
`misc`	Utility routines (deprecated)
`ndimage`	N-dimensional image processing and interpolation
`odr`	Orthogonal distance regression
`optimize`	Numerical optimization
`signal`	Signal processing
`sparse`	Sparse arrays, linear algebra and graph algorithms
`spatial`	Spatial data structures and algorithms
`special`	Special functions
`stats`	Statistical functions

CuPy

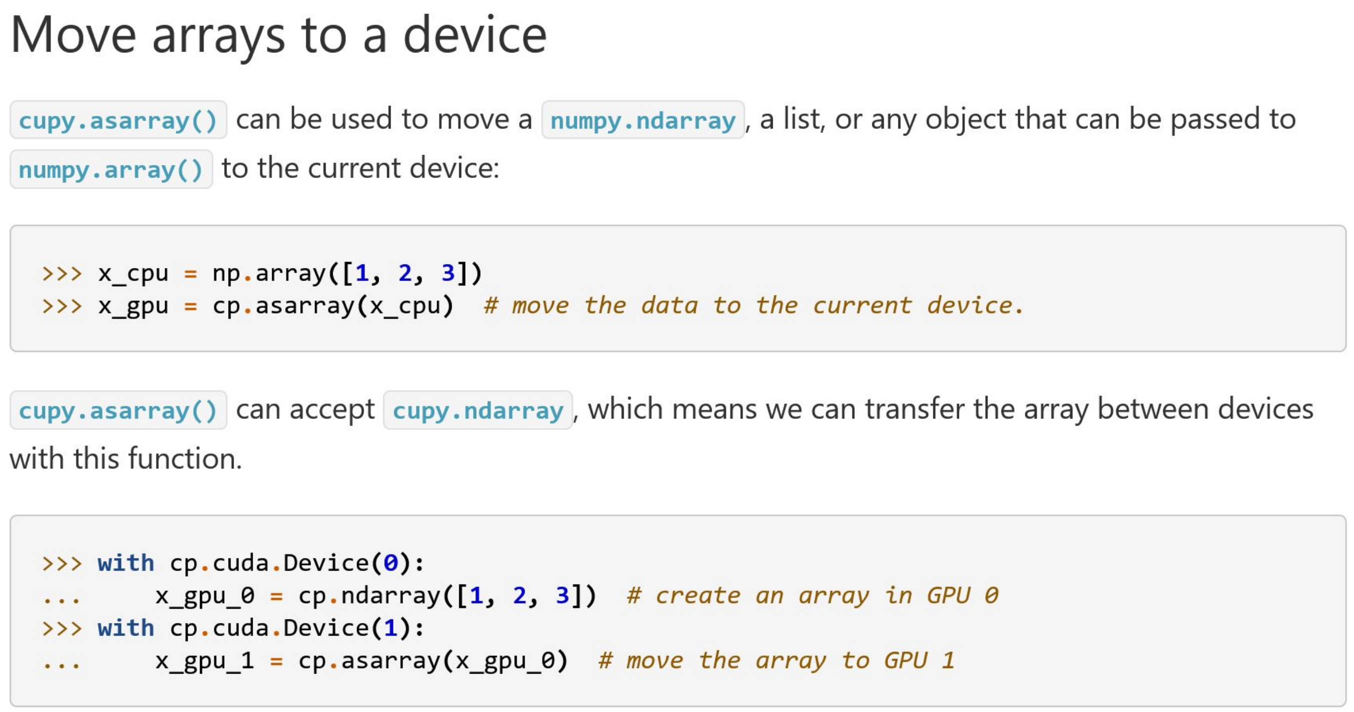
It is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python.
CuPy acts as a drop-in replacement to run existing NumPy/SciPy code on NVIDIA CUDA
or AMD ROCm platforms.

CuPy provides:



- **N-dimensional array** (`ndarray`): cupy.ndarray
  - Data types (dtypes): boolean (`bool_`), integer (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`), float (`float16`, `float32`, `float64`), and complex (`complex64`, `complex128`)
  - Supports the semantics identical to `numpy.ndarray`, including basic / advanced indexing and broadcasting
- **Sparse matrices**: cupyx.scipy.sparse
  - 2-D sparse matrix: `csr_matrix`, `coo_matrix`, `csc_matrix`, and `dia_matrix`
- **NumPy Routines**
  - Module-level Functions (`cupy.*`)
  - Linear Algebra Functions (`cupy.linalg.*`)
  - Fast Fourier Transform (`cupy.fft.*`)
  - Random Number Generator (`cupy.random.*`)
- **SciPy Routines**
  - Discrete Fourier Transforms (`cupyx.scipy.fft.*` and `cupyx.scipy.fftpack.*`)
  - Advanced Linear Algebra (`cupyx.scipy.linalg.*`)
  - Multidimensional Image Processing (`cupyx.scipy.ndimage.*`)
  - Sparse Matrices (`cupyx.scipy.sparse.*`)
  - Sparse Linear Algebra (`cupyx.scipy.sparse.linalg.*`)

Copying data from host (CPU) to device (GPU)



Move arrays to a device

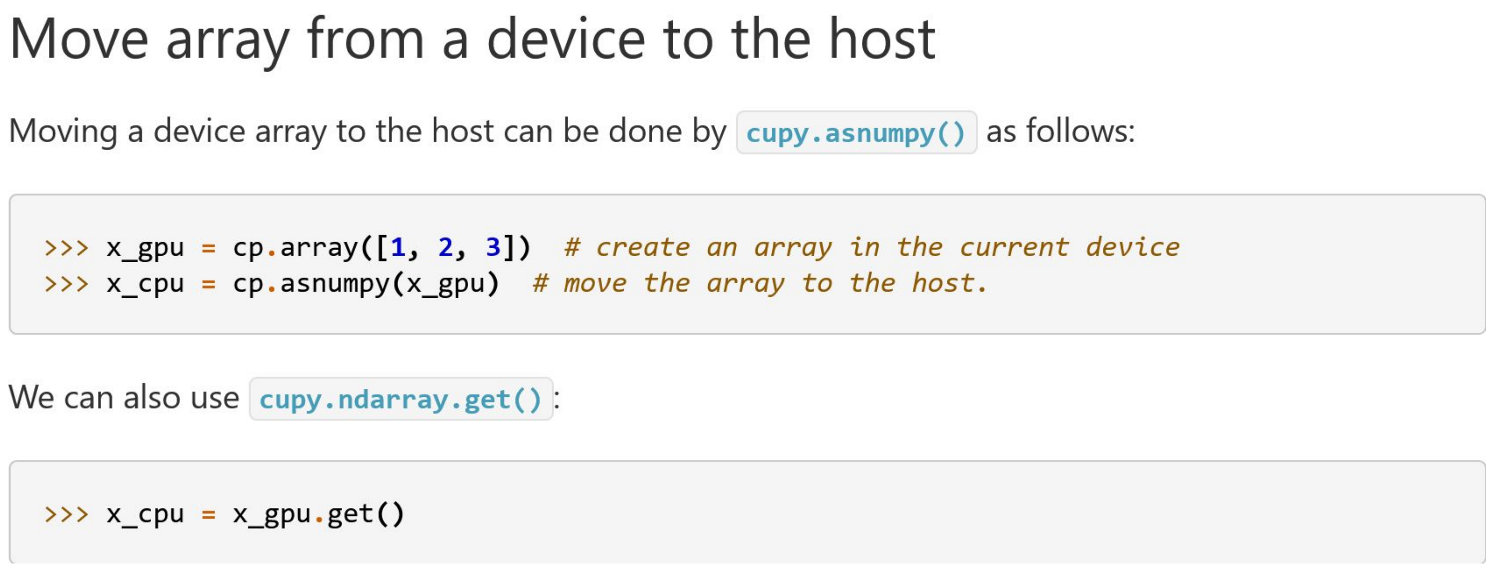
`cupy.asarray()` can be used to move a `numpy.ndarray`, a list, or any object that can be passed to `numpy.array()` to the current device:

```
>>> x_cpu = np.array([1, 2, 3])
>>> x_gpu = cp.asarray(x_cpu)  # move the data to the current device.
```

`cupy.asarray()` can accept `cupy.ndarray`, which means we can transfer the array between devices with this function.

```
>>> with cp.cuda.Device(0):
...     x_gpu_0 = cp.ndarray([1, 2, 3])  # create an array in GPU 0
>>> with cp.cuda.Device(1):
...     x_gpu_1 = cp.asarray(x_gpu_0)  # move the array to GPU 1
```

Copying data from device to host



Move array from a device to the host

Moving a device array to the host can be done by `cupy.asnumpy()` as follows:

```
>>> x_gpu = cp.array([1, 2, 3])  # create an array in the current device
>>> x_cpu = cp.asnumpy(x_gpu)  # move the array to the host.
```

We can also use `cupy.ndarray.get()`:

```
>>> x_cpu = x_gpu.get()
```

User-defined functions in Cupy

Users can define functions using pre-defined kernels types:

- raw kernels
- element-wise (templated or not)
- Reduce Kernels

Raw kernels in Cupy

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''', 'my_add')
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y))  # grid, block and arguments
>>> y
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

Examples of element-wise kernels

```
>>> squared_diff = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'squared_diff')
```

Time profiling Python

If you check out the built-in `time` module in Python, then you'll notice several functions that can measure time:

- `monotonic()`
- `perf_counter()`
- `process_time()`
- `time()`

Python 3.7 introduced several new functions, like `thread_time()`, as well as **nanosecond** versions of all the functions above, named with an `_ns` suffix. For example, `perf_counter_ns()` is the nanosecond version of `perf_counter()`

```python
# latest_tutorial.py

import time
from reader import feed

def main():
    """Print the latest tutorial from Real Python"""
    tic = time.perf_counter()
    tutorial = feed.get_article(0)
    toc = time.perf_counter()
    print(f"Downloaded the tutorial in {toc - tic:0.4f} seconds")

    print(tutorial)

if __name__ == "__main__":
    main()
```

Time profiling in Cupy

```
>>> import time
>>> start_gpu = cp.cuda.Event()
>>> end_gpu = cp.cuda.Event()
>>>
>>> start_gpu.record()
>>> start_cpu = time.perf_counter()
>>> out = my_func(a)
>>> end_cpu = time.perf_counter()
>>> end_gpu.record()
>>> end_gpu.synchronize()
>>> t_gpu = cp.cuda.get_elapsed_time(start_gpu, end_gpu)
>>> t_cpu = end_cpu - start_cpu
```

Matplotlib

Matplotlib is a 2D and 3D graphics library for generating scientific figures. Some of the advantages of this library include:

- Easy to get started - resembles IDL or MATLAB-stle procedure interface
- Support for LaTeX formatted labels and texts
- High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF
- Great control of every element in a figure, including figure size and DPI
- GUI for interactively exploring figures but supports figure generation without the GUI
- Integrates well with Jupyter notebooks

More information at the Matplotlib web page: http://matplotlib.org/