Python & NumPy Primer

IC-CVLab

2.3.2021
Python
About Python

• High-level, focus on **readable and concise code** - fast prototyping.

• **Interpreted** language, **dynamic typing** - very flexible.

• **Slow run-time** (dynamic typing, memory management, …)
  • Unless compiled to parallelized machined code (**Numba**)  

• Quick-and-dirty scripting, **glue code** - combine components written in other languages (NumPy, SciPy, scikit-learn).

• Vast amount of **libraries** - from backend web development, games to data science.
Working with Python

Installation

• Install the **interpreter** and **packages** manually
  • OSX and some Linux distributions already include Python
  • pip - package installer

• Get a readily available **distribution**
  • Anaconda

Running

• Terminal (interactive or script mode)

```
$ python
>>> print('hello CS322')
'hello CS322'
>>> # Use Ctrl-D, Ctrl-Z or quit() to exit
```

```
$ python my_script.py
'hello 'CS322'
```

• Jupyter Notebook
Data Types

- Python is dynamically typed language.
  - Data type inferred at runtime.
  - Can change during runtime.

```python
>>> a = 1
>>> print(type(a))
<class 'int'>

>>> a = 'python is fun'
>>> print(type(a))
<class 'str'>
```

- function `type()` returns the type of a value.

```python
>>> a = 42
>>> a = 3.142
>>> a = 3.
>>> a = 2+3j
>>> a = True
>>> a = None
```
Data Types

- **Strings** can be single, double or triple quoted.

```python
>>> a = 'stay strong'
>>> a = "it is not that difficult"

>>> a = '""This is a string that will span across multiple lines. Using newline characters and no spaces for the next lines.""'  # docstring
```

- Quotes do not change anything.

```python
>>> a = 'this is a string'
>>> b = "this is a string"
>>> print(b == a)
True
```
ADT (Abstract Data Type) - list

- Contains a series of values.

```
>>> a = []  # empty
>>> b = [1, 2, 3, 4]  # 4 elements
>>> c = [1, 'cat', 0.23]  # mixed types
>>> d = [1, ['cat', 'dog'], 2, 3]  # list in list
```

```
>>> a = ['great', 'minds', 'think', 'alike']
>>> print(a[0])  # zero indexed
'great'

>>> print(a[0:2])  # from start to end-1
['great', 'minds']

>>> a[2] = 'look'  # manipulate vals at given index
>>> print(a)
['great', 'minds', 'look', 'alike']
```
ADT - list

• Length of a list.

```
>>> a = ['great', 'minds', 'think', 'alike']
>>> print(len(a))  # length of list
4
```

• Adding item to a list.

```
>>> a = ['great', 'minds', 'think', 'alike']
>>> a.append('sometimes')
['great', 'minds', 'think', 'alike', 'sometimes']
```

• Extend a list by another list.

```
>>> a = ['great', 'minds']
>>> b = ['think', 'alike']
>>> c = a + b
['great', 'minds', 'think', 'alike']

>>> a.extend(b)  # in place
['great', 'minds', 'think', 'alike']
```
ADT - tuple

- Similar as list but fixed in size and **immutable** = change not allowed.
- Declaration with parentheses.

```python
>>> username = ('rick', 'morty', 'beth', 'squanchy')
>>> print(username[1])
'morty'

>>> username[0] = 'rick2'  # from other universe
TypeError: 'tuple' object does not support assignment
```

- **Tuple unpacking**

```python
>>> a, b = (1, 2)
>>> print(b)
2
```
ADT - dictionary

- Collection of key:value pairs
- Access value with keys
- Key must be immutable and unique

```python
>>> age = {'carl': 23, 'elis': 25}
>>> print(age['carl'])
23

>>> age['carl'] = 32  # modify
>>> print(age['carl'])
32

>>> age['peter'] = 18  # add new key
>>> print(age)
{'carl': 32, 'peter': 18, 'elis': 23}

>>> print(age.keys())  # get keys
dict_keys(['elis', 'carl', 'peter'])
>>> print(age.values())  # get values
dict_values([25, 23, 18])
```
Type Casting

```python
>>> a = 42  # int -> float
>>> a = float(a)
>>> print(a, type(a))
42.0 <class 'float'>

>>> a = 3.14  # float -> int (floor, not round!)
>>> a = int(a)
>>> print(a, type(a))
3 <class 'int'>

>>> a = '123'  # string -> int
>>> a = int(a)
>>> print(a, type(a))
123 <class 'int'>

>>> a = 123.14  # float -> string
>>> a = str(a)
>>> print(a, type(a))
'123.14' <class 'str'>
```
Type Casting

```python
>>> a = [42, 21.2, 'black']  # list -> tuple
>>> a = tuple(a)
>>> print(a)
(42, 21.2, 'black')

>>> a = (42, 21.2, 'black')  # tuple -> list
>>> a = list(a)
>>> print(a)
[42, 21.2, 'black']
```
Functions

```python
>>> def print_hello():
    print("hello world")

>>> def multiply(a, b):
    return a * b

>>> def add(a, b, c=0, d=1):
    return a + b + c + d

>>> add(1, 2, d=5)
8

• Default arguments and multiple return values.

```python
>>> def add(a, b, c=0, d=1):
    return a + b + c + d

>>> m, x = min_max([1, 2, 3, 4])
>>> print(m, x)
1, 4

• Multiple return values - a function returns a tuple, use tuple unpacking.

```python
>>> def min_max(l):
    return min(l), max(l)

>>> m, x = min_max([1, 2, 3, 4])
>>> print(m, x)
1, 4
```
Python comes with various useful **builtin functions**.

- range, zip, type, str, int, sorted, len, sum, max, min, abs, any, all, …

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

>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> list(zipped)
[(1, 4), (2, 5), (3, 6)]

>>> type('hello')
str
>>> sorted((2, 1, 3))
[1, 2, 3]
>>> sorted('eba')
['a', 'b', 'e']
```
Builtin Functions

```python
>>> len('abc')
3
>>> sum([1, 2, 3])
6
>>> max([1, 2, 3])
3
>>> min([1, 2, 3])
1
>>> abs(-2)
2
>>> a = [False, False, True]
>>> any(a)
True
>>> b = [False, False, False]
>>> any(b)
False
>>> c = [True, True, True]
>>> all(c)
True
```
List Comprehensions

- List comprehensions provide a concise way to create lists.
- Loosely follows mathematical set-builder notation.

\[ \{2^x \mid x \in \{0...10\}\} \]

```python
>>> powers2 = [2**x for x in range(11)]
>>> print(powers2)
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]
```

\[ \{2^x \mid x \in \{0...10\} \land x \text{ is even}\} \]

```python
>>> powers2ev = [2**x for x in range(11) if x % 2 == 0]
>>> print(powers2ev)
[1, 4, 16, 64, 256, 1024]
```
Import

• Gains access to code in another **module** by **importing** it.

```
>>> import numpy
>>> x = numpy.arange(5)
```

• Import module or just specified functions/variables/classes from the module.

```
>>> from numpy import arange
>>> x = arange(5)
```

• Select a name for imported module.

```
>>> import numpy as np
>>> x = np.arange(5)
```
• In Jupiter Notebook - function name with “?”.

```
>>> sum?
Signature: sum(iterable, start=0, /)
Docstring:
Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value. This function is intended specifically for use with numeric values and may reject non-numeric types.
Type:       builtin_function_or_method
```

• Read (Python and NumPy) documentation.
NumPy
About NumPy

• Core library for **scientific computing** in Python.

• High-performance multidimensional array - matrix operations.

• Wide ecosystem of libraries that take NumPy arrays as input.

**NumPy Arrays**

• high-performance multidimensional arrays.

• A grid of values, all of the **same type**.

• Indexed by a tuple of nonnegative integers.
  • Indexing syntax similar to lists, tuples, and dictionaries

• The **rank** of the array is the **number of dimensions**
# import numpy

```python
>>> import numpy as np

# create a rank 1 array
>>> np.array([1, 2, 3])
array([1, 2, 3])

# create a rank 2 array
>>> np.array([[1, 2, 3], [4, 5, 6]])
array([[1, 2, 3],
       [4, 5, 6]])
```
### Creating Arrays II

```python
>>> np.zeros((2,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])

>>> np.ones((2,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

>>> np.eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```
Creating Arrays III

```python
>>> np.arange(1, 7, 1).reshape((2, 3))
array([[1, 2, 3],
       [4, 5, 6]])

>>> np.linspace(1, 5, 9)
array([1., 1.5, 2., 2.5, 3., 3.5, 4., 4.5, 5.])
```
Inspecting Arrays

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

>>> x.shape
(2, 3)

>>> x.ndim
2

>>> x.size
6

>>> x.dtype
dtype('int64')

>>> y = np.array([1, 2, 3], dtype=np.float32)

>>> y.dtype
dtype('float32')
```
Indexing in Python is zero-based.

```python
>>> x = ['a', 'b', 'c', 'd', 'e']
>>> x[0]
'a'
```

The n-th last entry can be indexed by `-n`.

```python
>>> x[-1]
'e'
```

For a rank 2 array, the first index refers to the row & the second to the column.
Indexing multidimensional arrays

- A d-dimensional array - array of d−1 dimensional arrays
- A 2D numpy array is an array of 1D vectors (rows)

```python
>>> m = np.arange(9).reshape((3, 3))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])

>>> m[0]
array([0, 1, 2])

>>> m[1]
array([3, 4, 5])
```

- In the same way, a 3D array is an array of 2D matrices
Indexing multidimensional arrays

- NumPy also allows to index along a specific dimension explicitly

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

>>> print(A[0, 2])
3
```

- syntax:
  - comma ‘,’ - separate dimensions
  - colon ‘:’ - get all values in given dimension

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

>>> print(A[0, 2])
3
```
Indexing multidimensional arrays

- NumPy also allows to index along a specific dimension explicitly

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
\end{array}
\]

example: \(A[0, 2]\)
example: \(A[:, 1]\)

- syntax:
  - comma ‘,’ - separate dimensions
  - colon ‘:’ - get all values in given dimension

```python
>>> A = np.array([[1,2,3],[4,5,6]])
>>> print(A)
[[1 2 3]
 [4 5 6]]
>>> print(A[:, 1])  # all the rows, column 1
array([[2, 5]])
```
Indexing multidimensional arrays

• NumPy also allows to index along a specific dimension explicitly

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

```python
>>> print(A)
[[1 2 3]
 [4 5 6]]
```

```python
>>> print(A[1, :])  # all the columns, row 1
array([4, 5, 6])
```
Indexing multidimensional arrays

**Structural Indexing**

- Syntactic sugar used to add an extra dimension to an existing array.

```python
>>> x = np.array([1, 2, 3])
x.shape
(3,)

>>> x[np.newaxis].shape
(1, 3)

>>> x[:, np.newaxis].shape
(3, 1)

>>> y = np.arange(24).reshape((2, 3, 4))
>>> y.shape
(2, 3, 4)

>>> y[:, np.newaxis, :, :].shape
(2, 1, 3, 4)
```
Indexing multidimensional arrays

Structural Indexing

- Useful for broadcasting.
- Explicit shape, e.g. row/column vectors.

**Outer Product**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>18</td>
</tr>
</tbody>
</table>

**Inner Product (Dot Product)**

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

x1

x2
Indexing multidimensional arrays

Structural Indexing

```python
>>> x1 = np.array([[1, 2, 3]])

>>> x2 = np.array([[2, 4, 6]])

>>> x1 = x1[:, np.newaxis]
>>> x1.shape
(3, 1)

>>> x2 = x2[np.newaxis]
>>> x2.shape
(1, 3)

>>> x1 @ x2
array([[2, 4, 6],
       [4, 8, 12],
       [6, 12, 18]])
```

outer product
Indexing multidimensional arrays

Inner product (dot product)

```python
>>> x1 = np.array([[1, 2, 3]])
>>> x2 = np.array([[2, 4, 6]])

>>> x1 = x1[np.newaxis]
>>> x1.shape
(1, 3)

>>> x2 = x2[:, np.newaxis]
>>> x2.shape
(3, 1)

>>> x1 @ x2
array([[28]])
```
Slicing: Basics

• Slicing is indexing of the form `[start : stop : step]`
  • start including
  • stop excluding

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

>>> x[1:10:2]
array([ 2,  4,  6,  8, 10])

• If not specified, `start = 0`, `stop = last elem.`, `step = 1`

```python
>>> x[::2]
array([ 1,  3,  5,  7,  9])

>>> x[9:0:-1]
array([10,  9,  8,  7,  6,  5,  4,  3,  2])

>>> x[:::-1]
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```
Slicing: Basics

- Slicing provides a **view** of the original array.

```python
>>> y = x[0:2]
>>> y[:] = 42
>>> x
array([42, 42,  3,  4,  5,  6,  7,  8,  9, 10])
```
Slicing: Example

Syntax: `[start : stop : step]`

```python
>>> a[0,3:5]
array([[3,4]])
```

```python
>>> a[4:,4:]
array([[44, 45],
       [54, 55]])
```

```python
>>> a[:,2]
array([2,12,22,32,42,52])
```

```python
>>> a[2::2,::2]
array([[20,22,24],
       [40,42,44]])
```
Masking: Basics

• Masking is indexing an array with an identically shaped Boolean array.

• Elements indexed with True / False are taken / discarded, respectively.

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

>>> mask = np.array([True, False, False, True, False])
>>> x[mask]
array([1, 4])
```
Masking: Example

- Get all even numbers in [1, 6] which are multiplies of 3.

```python
>>> m = np.arange(1, 7)
aarray([1, 2, 3, 4, 5, 6])

>>> div2 = (m % 2 == 0)
aarray([False, True, False, True, False, True])

>>> div3 = (m % 3 == 0)
aarray([False, False, True, False, False, True])

>>> mask = div2 & div3
aarray([False, False, False, False, False, True])

>>> m[mask]
aarray([6])
```
Type casting

- Use `.astype` to convert numpy arrays between types.
- Use `dtype` to check the type.

```python
>>> m = np.arange(0, 5)
array([ 0, 1, 2, 3, 4])

>>> m.astype(np.float32)
array([ 0., 1., 2., 3., 4.])

>>> m.astype(np.bool)
array([False, True, True, True, True])

>>> m.dtype
dtype('bool')
```
Broadcasting

• Describes how numpy treats arrays with different shapes during arithmetic operations.

• **Fast** - operation is **vectorised**, heavily **parallelized**.

• Performed **without making needless copies** of data.
How broadcast works

- Smaller array is “broadcast” across the larger array so that they have compatible shapes
Broadcasting: Basic Example

• Add scalar to every element of a vector.

```python
>>> my_vector = np.array([1, 3, 5, 7, 9])
>>> my_scalar = -5
>>> print(my_vector)
[1 3 5 7 9]

>>> new_vector = my_vector + my_scalar
>>> print(new_vector)
[-4 -2 0 2 4]
```
Broadcasting: Advanced Example

• Convert binary numbers (in rows) to decimal
Broadcasting Rules

- The corresponding dimensions of 2 arrays must satisfy one of the following:
  - Same dimension
  - One of the dimensions is 1
- Non-existent dimensions are treated as 1

<table>
<thead>
<tr>
<th>Array</th>
<th>Dimensions</th>
<th>Result</th>
<th>Dimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(2d array): 5 x 4</td>
<td></td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>(1d array): 1</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>Result</td>
<td>(2d array): 5 x 4</td>
<td></td>
<td>Result</td>
</tr>
</tbody>
</table>

A (3d array): 15 x 3 x 5
B (2d array): 3 x 1
Result (3d array): 15 x 3 x 5

```python
>>> a = np.ones((15, 3, 5))
>>> b = np.arange(3).reshape((3, 1))
>>> c = a * b
>>> c.shape
(15, 3, 5)
```
Common Functions in Numpy

\[ a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \]

- When we do not specify the \textbf{axis} parameter:

```python
>>> sum_all_vals = np.sum(a)
>>> print(sum_all_vals)
21
```
Common Functions in Numpy

• Sum along the columns:

```
>>> sum_along_cols = np.sum(a, axis=1)
>>> print(sum_along_cols)
[ 6 15 ]
```

```
1 2 3
4 5 6
```

```
[123]
[456]
```

```
[6]
[15]
```

• Sum along the rows:

```
>>> sum_along_rows = np.sum(a, axis=0)
>>> print(sum_along_rows)
[ 5 7 9 ]
```

```
[1 2 3]
[4 5 6]
```

```
[5 7 9]
```

```
[123]
[456]
```

```
[579]
```

```
[615]
```
Common Functions in Numpy

\[ a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \]

- Find minimum value in an array:
  ```python
  >>> all_min = np.min(a)
  >>> print(all_min)
  [ 1 ]
  ```

- Along cols:
  ```python
  >>> col_min = np.min(a, axis=1)
  >>> print(col_min)
  [ 1 4 ]
  ```

- Along rows:
  ```python
  >>> row_min = np.min(a, axis=0)
  >>> print(row_min)
  [ 1 2 3 ]
  ```

For finding the maximum value, you can use `np.max`
Common Functions in Numpy

• Find **index** of minimum value in an array:

\[ b = \begin{bmatrix} 5 & 3 & 10 \end{bmatrix} \]

```python
>>> b = np.array([5, 3, 10])
>>> min_ind_b = np.argmin(b)
>>> print(min_ind_b)
1
```

• Be careful with multi-dimensional arrays!

\[ c = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 6 \end{bmatrix} \]

```python
>>> c = np.array([[1, 2, 3], [4, 0, 6]])
>>> min_ind_c = np.argmin(c)
>>> print(min_ind_c)
4
```

• To get 2D index, use `np.unravel_index()`
Common Functions in Numpy

\[ a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \]

• Find mean value of an array:

```python
>>> overall_mean = np.mean(a)
>>> print(overall_mean)
3.5
```

• Along columns:

```python
>>> col_mean = np.mean(a, axis=1)
>>> print(col_mean)
[2. 5.]
```

• Along rows (this is useful for whitening your data!):

```python
>>> row_mean = np.mean(a, axis=0)
>>> print(row_mean)
[2.5 3.5 4.5]
```

You can use `np.std` similarly to find the standard deviation and `np.var` to find the variance
Random

• The module `np.random` implements pseudo-random number generators for various distributions.

• Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range `[0.0, 1.0)`.

```python
>>> x = np.random.random()
>>> print(x)
0.22402776143655379
```
Random

• How to fix the pseudo-random generator?

```python
>>> np.random.random()
0.4353
>>> np.random.random()
0.4204
```

• Use `seed`.

```python
>>> np.random.seed(2)
>>> np.random.random()
0.4360
>>> np.random.seed(2)
>>> np.random.random()
0.4360
```
Random: Useful Functions

- Getting random integer \( n \) such that \( a \leq n \leq b \):

```python
>>> n = random.randint(3, 10)
>>> print(n)
7
```

- Getting random float \( x \) such that \( \text{min} \leq x \leq \text{max} \):

```python
>>> minim = 3
>>> maxim = 10
>>> x = (maxim - minim) * random.random() + minim
>>> print(x)
3.1815
```
Random: Useful Stuff

• How to shuffle (in-place) randomly list of indices?

```python
>>> x = np.arange(10)
>>> print(x)
[0 1 2 3 4 5 6 7 8 9]

>>> np.random.shuffle(x)
>>> print(x)
[9 5 4 7 1 8 0 3 2 6]
```

• Some useful distributions (uniform, normal):

```python
>>> x = random.uniform(low=0, high=1, size=(2, 3))
>>> x = random.randn(2, 3)
```
References

- Python Documentation: https://docs.python.org/3/
- Official python3 tutorial: https://docs.python.org/3/tutorial/index.html
- All built-in functions: https://docs.python.org/3/library/functions.html
- NumPy Basics: https://docs.scipy.org/doc/numpy/user/basics.html
- Official NumPy tutorial: https://docs.scipy.org/doc/numpy/user/quickstart.html