Watch by Tuesday, October 20, 2020 | **Lesson #5**

NumPy

Arrays and functions

OCEAN 215 | Autumn 2020

Ethan Campbell and Katy Christensen

What we'll cover in this lesson

1. **Functions and arguments**
2. NumPy arrays – arithmetic, logical operations, indexing
3. NumPy functions and constants

Functions and arguments

Function name — An "argument" or "parameter"

```
len([6,8,7,5])
```

The function "returns" or "evaluates to" the integer 4

Some functions act on a target

The "target" of the function

Function name

```
'python'.upper()
```

The function returns `'PYTHON'`

Values returned by functions can be stored in a variable

The "target" of the function

Function name

'python'.**upper()**

new_string = 'python'.**upper()**

Some functions don't return anything

```
numbers = [6,8,7,5]
```

```
numbers.sort()
```

This function returns nothing at all!

It simply modifies `numbers` "in-place," which becomes `[5,6,7,8]`.

Some functions have named arguments

"Named" or "keyword" argument → `reverse`

Argument value → `True`

```
numbers.sort(reverse=True)
```

Now, `numbers` will be sorted in reverse order: `[8,7,6,5]`.

Named arguments always have a default value

The “default” value of `reverse`

```
numbers.sort(reverse=False)
```

is equivalent to:

```
numbers.sort()
```

Functions can have **both** positional and named arguments

“Positional” arguments have a fixed order

```
function_name(arg1,arg2,arg3,…,named_arg1=default1,
named_arg2=default2,…)
```

“Named” arguments can be provided in any order,
but they must follow any positional arguments

What we'll cover in this lesson

1. Functions and arguments
2. **NumPy arrays – arithmetic, logical operations, indexing**
3. NumPy functions and constants

Loading NumPy ("Numeric Python")

Makes this package available to Python

This is a shortcut; you can choose any name but `np` is most common

```
import numpy as np
```

This part is technically optional

Package names are usually all lowercase

Checking a package's version

```python
import numpy as np

print(np.__version__)
```

```
1.18.5
```

That's a double underscore: __

The NumPy array (`ndarray`)

"N-dimensional array" (e.g. 1-D, 2-D, 3-D, 4-D, etc.)

```
np.array([5,6,7,8])
```

Similarities between lists and NumPy 1-D arrays

Both are **mutable** (they can be changed)

```
numbers = np.array([5,6,7,8])
numbers[1] = 13
print(numbers)
```

```
[ 5 13  7  8]
```

Both are **iterable**

```
for num in numbers:
  print(num)
```

```
5
13
7
8
```

Both are compatible with **indexing** and **slicing**

```
print(numbers[-3:])
```

```
[13  7  8]
```

Find length using **`len()`**

```
print(len(numbers))
```

```
4
```

Check membership using **`in`** and **`not in`**

```
print(13 in numbers)
print(14 in numbers)
```

```
True
False
```

Differences between lists and NumPy 1-D arrays

Lists	**NumPy 1-D arrays**
• Lists can contain a mix of object types (integers, strings, sub-lists, etc.)	• Arrays can contain only a single object type (check using `.dtype`, change using `.astype()`)
• Lists are computationally inefficient (avoid using to store large data sets)	• Arrays are fast for computation and small in memory (great for big data)

```python
numbers = np.array([5,6,7,8])
print(numbers.dtype)
print(numbers.astype(str))
```

```
int64
['5' '6' '7' '8']
```

Differences between lists and NumPy 1-D arrays

Lists

- Lists don't preserve scientific notation in floating-point numbers

```
print([3.5e9,1.4e-3])
```

```
[3500000000.0, 0.0014]
```

- Use Python's **in-place** `append()` or `extend()`, `insert()`, `del`, `reverse()`, `remove()`, `pop()`

```
numbers = [5,6,7,8]
numbers.append([9,10])
print(numbers)
```

```
[5, 6, 7, 8, [9, 10]]
```

NumPy 1-D arrays

- Arrays preserve scientific notation

```
print(np.array([3.5e9,1.4e-3]))
```

```
[3.5e+09 1.4e-03]
```

- NumPy's `append()`, `insert()`, `delete()`, `flip()` functions are **not in-place**; note the different syntax; no functions to remove, pop

```
numbers = np.array([5,6,7,8])
numbers = np.append(numbers,[9,10])
print(numbers)
```

```
[ 5  6  7  8  9 10]
```

Differences between lists and NumPy 1-D arrays

Lists	**NumPy 1-D arrays**

Lists

- Convert from list → array using:

```
my_list = [5,6,7,8]
my_array = np.array(my_list)
```

- Adding lists **concatenates** (joins) **them**:

```
a = [1,2,3,4]
b = [5,6,7,8]
print(a + b)
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

NumPy 1-D arrays

- Convert from array → list using:

```
my_list1 = my_array.tolist()
my_list2 = list(my_array)
```

- Adding arrays actually **adds them**!*

```
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])
print(a + b)
```

```
[ 6  8 10 12]
```

* Note that NumPy also has a `concatenate()` function.

Arithmetic operations with arrays

Arithmetic operators

+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponential
%	Remainder
//	Floor

Element-wise arithmetic between two or more arrays

```python
a = np.array([1,2,3,4])
b = np.array([5,6,7,8])

print('a + b =',a + b)
print('a - b =',a - b)
print('a * b =',a * b)
```

```
a + b = [ 6  8 10 12]
a - b = [-4 -4 -4 -4]
a * b = [ 5 12 21 32]
```

Element-wise arithmetic with an array and a number

```python
print('a + 10 =',a + 10)
print('10 * a =',10 * a)
print('a / 10 =',a / 10)
print('a**2 =',a**2)
```

```
a + 10 = [11 12 13 14]
10 * a = [10 20 30 40]
a / 10 = [0.1 0.2 0.3 0.4]
a**2 = [ 1  4  9 16]
```

Element-wise operations require arrays to be the same dimensions

```
x = np.array([1,2,3])
y = np.array([11,12,13,14,15])

print(x + y)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-97-d5d99ad6233b> in <module>()
      2 y = np.array([11,12,13,14,15])
      3
----> 4 print(x + y)

ValueError: operands could not be broadcast together with shapes (3,) (5,)
```

Logical operations with arrays

Comparison operators

==	Equal
!=	Not equal
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

Element-wise comparisons between two arrays or an array and a number

```
u = np.array([1,2,3,4])
v = np.array([0,2,4,6])

print(u == v)
print(u < v)
print(v != 0)
print(v <= 4)
```

```
[False  True False False]
[False False  True  True]
[False  True  True  True]
[ True  True  True False]
```

Instead of comparing Boolean arrays with `and`/`or`, use `np.logical_and()` and `np.logical_or()`

```
bool1 = np.array([True,False,True])
bool2 = np.array([True,False,False])

print(np.logical_and(bool1,bool2))
print(np.logical_or(bool1,bool2))
```

```
[ True False False]
[ True False  True]
```

New indexing options with arrays

When you want to access certain value(s) in an array:

```python
v = np.array([10,11,12,13])

print(v[3])

print(v[[2,3]])

print(v[v >= 12])

print(v[[False,False,True,True]])
```

Python prints:

`13`	Traditional list-style **single index**
`[12 13]`	**Multiple indices** retrieves multiple elements
`[12 13]`	**Logical conditions** also work…
`[12 13]`	… because they evaluate to **Boolean arrays**

When you want the indices of certain values in an array:

```python
print(np.where(v >= 12))

print(np.where(v >= 12)[0])
```

`(array([2, 3]),)`	`np.where()` gives the indices at which a Boolean condition is satisfied…
`[2 3]`	… but you have to index into the result using `[0]`

What we'll cover in this lesson

1. Functions and arguments
2. NumPy arrays – arithmetic, logical operations, indexing
3. **NumPy functions and constants**

Most functions acting on NumPy arrays can be called two ways

```
x = np.array([10,11,12,13])
```

```
np.sum(x)
```

← Evaluates to: 46

```
x.sum()
```

← Evaluates to: 46

NumPy functions can also be applied to lists

```
x = [10,11,12,13]
```

```
np.sum(x)
```

← Evaluates to: 46

```
x.sum()
```

← Evaluates to: 46

Mathematical reductions (array → number)

```
x = np.array([10,11,12,13])
```

Function:	Purpose:	Evaluates to:
`np.sum(x)`	Sum	46
`np.mean(x)`	Mean (average)	11.5
`np.median(x)`	Median	11.5
`np.max(x)`	Maximum value	13
`np.min(x)`	Minimum value	10
`np.std(x)`	Standard deviation	1.11803...

Mathematical constants (each return a `float`)

Constant value:	**Purpose:**	**Evaluates to:**
`np.pi`	π (pi)	`3.14159`...
`np.e`	*e* (Euler's number)	`2.71828`...
`np.inf`	Positive infinity	`inf`
`np.nan`	"Not a Number" (used as a placeholder for missing data)	`nan`

Note:

```python
print(5 * np.inf)
print(5 * np.nan)
```

```
inf
nan
```

Element-wise functions (number → number, or array → array)

Function:	Purpose:	Evaluates to arrays:
`np.absolute([-2,-1])`	Absolute value	`[2,1]`
`np.round([5.23,5.29],1)`	Round to a certain decimal place	`[5.2,5.3]`
`np.sqrt([4,9,16])`	Square root (same as `**0.5`)	`[2.,3.,4.]`
`np.exp([0,1,2])`	Exponential (same as `np.e**`)	`[1.,2.718…,7.389…]`
`np.sin([0,np.pi/2])`	Sine (from radians)	`[0.,1.]`
`np.cos([np.pi,2*np.pi])`	Cosine	`[-1.,1.]`

Functions to create new arrays

Function:	Purpose:	Evaluates to arrays:
`np.zeros(4)`	Array of given length filled with zeros	`[0.,0.,0.,0.]`
`np.ones(4)`	Array of given length filled with ones	`[1.,1.,1.,1.]`
`np.full(4,2)`	Array of given length filled with given value	`[2,2,2,2]`
`np.arange(4)`	Same as `range()`…	`[0,1,2,3]`
`np.arange(0,1,0.25)`	…except floats and fractional increments are allowed	`[0.,0.25,0.5,0.75]`
`np.linspace(0,1,5)`	Returns the given number of evenly spaced values from start to end (both are inclusive)	`[0.,0.25,0.5,0.75,1.]`