# Python 06

**Elias Wachmann**

2024

# Content

1. Indexing revisited
2. Scope
3. Call by value vs. call by reference
4. Copy vs. Deepcopy
5. Further numpy examples

# Indexing revisited

# Logical indexing

Using logical indexing, we can select elements of an array that satisfy a certain condition. For example, we can select all elements of an array that are larger than a certain value. The result is a 1D array containing the selected elements.

```
1 M = np.reshape(np.arange(6), (2, 3))
2 even_num = M % 2 == 0
3 bigger_than_2 = M > 2
4 prime_num = np.logical_or(M == 2, M == 3)
5 print(M[even_num])  # [0 2 4]
6 print(M[bigger_than_2])  # [3 4 5]
7 print(M[prime_num])  # [2 3]
```

# Logical Operators on numpy arrays

Numpy arrays can be combined using logical operators:

- `logical_and`
- `logical_or`
- `logical_not`
- `logical_xor`

They evaluate the truth tables (given in Python 05) element-wise.

# Logical Operators on numpy arrays

Examples using logical operators:

```
1 A = np.array([[True, False, False], [False,
     True, True]])
2 B = np.array([[True, False, True]])
3 res = np.logical_and(A, B)
4 same = A & B  # or really the same?
5 or_ = A | B
6 not_1 = np.logical_not(B)  # output?
7 not_2 = ~B  # any different?
8 xor = A ^ B
```

**Note:** B is evaluated on both rows of A.

# Logical Operators on numpy arrays caveats

```
1 res = np.logical_and(A, B)
```

Output:

```
1 # output: [[ True False False]
2 #          [False False  True]]
```

# np.logical_and/or/not() vs. & / | / ~

While the `logical_and/or/not` operators evaluates the truth table element-wise, the binary `& / | / ~` operators evaluates the truth table bitwise.

**But why should I care?**

Maybe you want to check which elements are not equal to zero between both arrays:

```
1 A = np.array([42, 15, 0])
2 B = np.array([-51, 0, 15])
3 res = np.logical_and(A, B)
4 same = A & B  # or really the same?
```

# np.logical_and/or/not() vs. & / | / ~

## But why should I care?

```
A = np.array([42, 15, 0])
B = np.array([-51, 0, 15])
res = np.logical_and(A, B)
same = A & B  # or really the same?
print(res)  # [True False  True]
print(same)  # [8   0  0]
```

Remember that the **bitwise** operators evaluate the truth table **bitwise**. Therefore, the result is not what we want (or maybe expected).

# np.logical_and/or/not() vs. & / | / ~

## But why should I care?

```
1 A = np.array([42, 15, 0])
2 B = np.array([-51, 0, 15])
3 res = np.logical_and(A, B)
4 same = A & B  # or really the same?
5 print(res)  # [True False  True]
6 print(same)  # [8   0  0]
```

| | | | | | | |
|---|---|---|---|---|---|---|
| | 42: | 0 | 0 | 1 | 0 | 1 | 0 |
| AND | -51: | 1 | 1 | 0 | 0 | 1 | 0 |
| | 8: | 0 | 0 | 0 | 0 | 1 | 0 |

# Scope

# Scopes in python

Variables in python have a scope that defines where they can be accessed.

```
1  global_ = 42
2
3
4  def f():
5      local_ = 43
6      print('local_:', local_)
7      print('global_:', global_)
8
9
10 print('global_:', global_)
11 print('local_:', local_)  # NameError
```

# Scopes in python

A scope is always limited to the current block (/indent).
Variables declared in the outermost scope are called **global** and can be accessed from anywhere.

Variables declared in a function are called **local** and can only be accessed from within the function.

In general, it is good practice to **avoid** global variables!

# Call by value vs. call by reference

# Call by value vs. call by reference

Parameter can be given to functions by value or by reference.

Primitive types – such as `int`, `str` and `float` – are passed by value (a separate copy is available).

Mutable types – such as `lists`, `dictionaries` or `objects` in general are passed by reference (they refer back to the original).

# Call by value vs. call by reference

```
1  def callbyvalue(x):
2      x = x + 1
3
4  def callbyreference(x):
5      x[0] = 42
6
7  myvar = 11
8  myarr = [1, 2, 3]
9  callbyvalue(myvar)
10 print("myvar =", myvar)   # 11
11 callbyreference(myarr)
12 print("myarr =", myarr)   # [42, 2, 3]
```

# Copy vs. Deepcopy

# What is a copy?

A copy is a new object that is created from an existing object.

**Shallow copy:** A new object is created, but the elements of the new object are references to the elements of the original object.

**Deep copy:** A new object is created, and the elements of the new object are copies of the elements of the original object.

# Create a shallow copy (copy)

Using pythons build-in copy module, we can create a shallow copy of an object.

```
1 original_list = [1, 2, 3, [4, 5, 6]]
2 shallow_copy1 = original_list.copy()
3 shallow_copy2 = original_list[:] # same as
    copy()
4 shallow_copy1[3][0] = 42
5 shallow_copy2[1] = 101
6
7 print(original_list) # [1, 2, 3, [42, 5, 6]]
8 print(shallow_copy1) # [1, 2, 3, [42, 5, 6]]
9 print(shallow_copy2) # [1, 101, 3, [42, 5,
    6]]
```

# Create a (deep)copy (np.copy)

The whole object can be copied using the copy function.

```
A = np.array([1,2,3,4])
B = np.copy(A)
C = A # only a reference to A
B[0] = 42
print(A)   # [1 2 3 4]
print(B)   # [42 2 3 4]
print(C)   # [1 2 3 4]
```

# Further numpy examples

# Numpy examples – Reshaping

reshape can be used to change the shape of an array.

```
square_matrix = np.array([[1, 2], [3, 4]])
column = square_matrix.reshape(4, 1)
row = square_matrix.reshape(1, 4)
print(column)
# [[1]
#  [2]
#  [3]
#  [4]]
print(row)  # [[1, 2, 3, 4]]
```

# Numpy examples – Ravel

ravel can be used to flatten an array.

```
square_matrix = np.array([[1, 2], [3, 4]])
print(square_matrix)
# [[1 2]
# [3 4]]
ravelled = square_matrix.ravel()
print(ravelled)  # [1 2 3 4]
```

flatten is another function that can be used to flatten an array.