

# 1 Wstęp

Cześć! Skoro tutaj trafiłeś, to najprawdopodobniej należysz do jednej z dwóch grup: albo zastanawiasz się, w jaki sposób można przetwarzać obrazy cyfrowe, albo: co jeszcze można zrobić w Pythonie? Mamy nadzieję, że w naszym poradniku znajdziesz odpowiedź. A nawet jeśli wpadłeś na niego zupełnie przypadkowo, to i tak lektura może okazać się całkiem interesująca ;)

Poniższy przewodnik jest wynikiem pracy trzech studentów w ramach projektu zaliczeniowego z przedmiotu związanego z cyfrowym przetwarzaniem obrazów. Mamy nadzieję, że zagadnienia w nim przedstawione, zachęcą Cię do dalszego rozwoju w tym kierunku :)

## Dlaczego Python?

Wybraliśmy ten język, ponieważ obecnie jest to jeden z najpopularniejszych języków programowania. Dodatkowo ma prostą składnię i należy do grupy języków wysoiego poziomu – czyli stosujemy w nim szereg gotowych funkcji, które ułatwiają pracę! Dzięki temu powstało wiele bibliotek dedykowanych różnym problemom/zadaniom, a Python stał się bardzo wszechstronnym narzędziem.

I śmieszy nas Monty Python (to od tej grupy komików wywodzi się nazwa języka) – ale spokojnie, w poradniku skupimy się na rozwiązywaniu powszechnych zadań związanych z przetwarzaniem obrazów cyfrowych, także zapraszamy do lektury, nawet jeśli masz zupełnie inne poczucie humoru.

## Python 2.x a 3.x

### Ramy czasowe

Python 2.x jest z nami od początku XXI wieku. Wprowadził takie cechy jak list comprehension (`[str(x) for x in range(10)]`), jednak jego wsparcie zostanie porzucone od 2020 roku. Oznacza to, że poza nielicznymi przypadkami (rozwijanie starych projektów, których konwersja do 3.x może okazać się zbyt ryzykowna i/lub pracochłonna) nie warto rozważać, czy wybrać Pythona 2.x czy 3.x.

Python 3. został wydany pod koniec 2008 roku. Obecnie wykorzystuje się jego wersje 3.6 (współcześnie 99% bibliotek) oraz 3.7 (ostatnia stabilna wersja, trwają prace nad wsparciem brakujących bibliotek).

Z tych względów w tym poradniku wykorzystywana będzie wersja 3.6.

### Różnice

Jeśli do tej pory korzystałeś tylko z wersji 2.x, polecamy zapoznać się z artykułem: [https://sebastianraschka.com/Articles/2014\\_1](https://sebastianraschka.com/Articles/2014_1), gdzie opisane zostały główne różnice.

## Podstawowe komendy

**Poniżej znajdziesz przykładowe pętle. Zauważ, że bloki kodu nie są wydzielone żadnym widocznym znakiem czy słowem (pętle nie kończą się klamrą, nawiasem lub "endem") – w Pythonie wewnętrze pętli (lub funkcji) zaznaczamy poprzez odpowiednie wcięcie (tabulator). Dzięki temu kod jest przejrzysty i łatwy do odczytania. Jeśli jednak po zakończeniu pętli zapomnisz wrócić kursorem do początku linii, kolejne instrukcje będą wykonywane jako dalsza część danego bloku, co może dać nieoczekiwane efekty! Lepiej o tym pamiętać.**

- wyrażenia warunkowe

```
python_is_easy = True # boolean True lub False
if python_is_easy:
    print('Python is easy!')
```

## 1 Wstęp

```
else:  
    print('Python is hard :(')  
Python is easy!  
• pętla while  
  
string = r'W pythonie możemy używać pojedynczych lub podwójnych apostrofów do znaków bądź słów'  
słowa = string.split(' ') # ta metoda dzieli nam ciąg znaków/słów wg podanego kryterium, tutaj po spacjach.  
licznik = 0  
  
while licznik<len(słowa):  
    print(słowa[licznik])  
    licznik+=1  
  
W  
pythonie  
możemy  
używać  
pojedynczych  
lub  
podwójnych  
apostrofów  
do  
znaków  
bądź  
słów
```

**WAŻNE!** W tym języku indeksujemy od 0.

- pętla for (wraz z uruchamianiem paczek)

```
import tqdm # ta paczka pozwoli nam na wygenerowanie paska postępu (przydatne przy długich operacjach)  
import urllib # pozwoli nam na przeczytanie pliku tekstowego z internetu  
  
with urllib.request.urlopen('http://files.catwell.info/misc/mirror/zen-of-python.txt') as txt:  
    for line in tqdm.tqdm(txt):  
        print(line.decode("utf-8"))  
  
21it [00:00, 35203.99it/s]
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Jest to znane Zem of Python. Możemy też uzyskać je poprzez użycie komendy *import this*.

Jeśli korzystasz z Jupyter Notebook komendy możemy wywoływać również z poziomu terminala/shell – należy po prostu poprzedzić komendę wykrzyknikiem. Kolejne polecenia możemy rozdzielać przecinkiem.

```
! python -c "import antigravity"
```

Wykorzystaliśmy tutaj tzw. *list comprehension*, które umożliwia uzyskanie listy z elementami przetworzonymi lub przefiltrowanymi wybranym przez nas kryterium. Wykorzystywane często przy prostych operacjach. Działają też dla obiektów tuple (krotka) lub dictionary (słownik).

## Instalacja bibliotek

Jak wspomnieliśmy, Python posiada wiele bibliotek dedykowanych do różnych celów. Aby móc z nich skorzystać, należy je najpierw aktywować poleceniem `import`. Część podstawowych modułów jest wbudowana, inne trzeba zainstalować. Istnieje wiele sposobów, by to zrobić, oto przykładowe:

- pojedynczo z poziomu konsoli

```
! pip install pandas -q
```

- “hurtowo” wykorzystując plik requirements.txt, który jest najprostszą metodą przechowywania niezbędnych paczek dla danego projektu (tutaj widoczne są biblioteki wykorzystywane przez nas)

```
with open('../requirements.txt', 'r') as f:
    for req in f.readlines():
        print(req)
```

jupyter

numpy

pillow

matplotlib

tqdm

opencv-python<4

scipy

numba

```
! pip install -r ../requirements.txt -q
```

Resztę potrzebnych komend poznasz w kolejnych rozdziałach poradnika. To jak, zaczynamy?



## 2 Początki: odczyt i zapis obrazu

Aby móc przetwarzać obraz, najpierw należy wprowadzić go do środowiska. W zależności od użytej biblioteki, polecenie będzie inne, dlatego zebraliśmy je w jednym miejscu.

Pamiętaj, by na początku załadować wymagane pakiety. Warto od razu nadać im własne, krótsze nazwy, ponieważ używając danej funkcji, należy wskazać, z jakiej biblioteki ona pochodzi.

```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import cv2
```

### Wczytanie obrazu

#### lokalnego

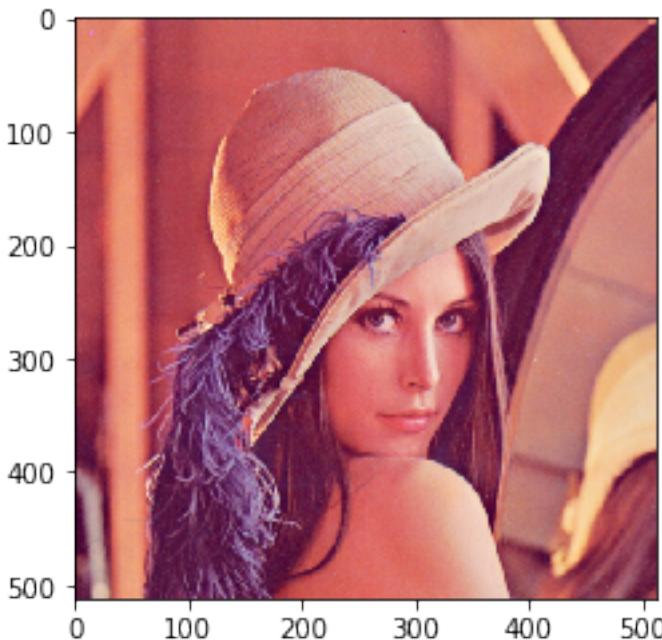
- biblioteka Pillow

```
img_pil = Image.open('..../obrazy_testowe/lena_512x512.png')
img_pil.resize((200,200))
```



- biblioteka matplotlib

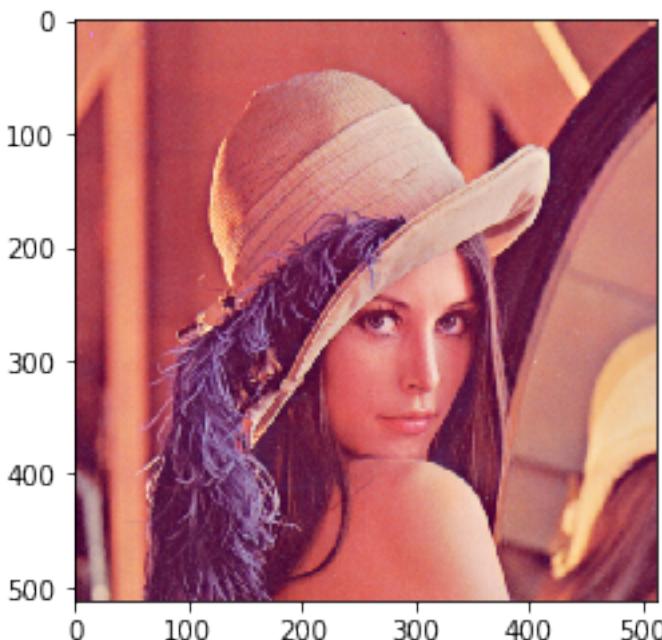
```
img_plt = plt.imread('..../obrazy_testowe/lena_512x512.png')
plt.imshow(img_plt)
plt.show()
```



Biblioteka ta działa podobnie jak Matlab; znajdziemy w niej też funkcje `imread()` oraz `imshow()`. Należy jednak dodać komendę `show()`, aby przedstawić obrazek. Wcześniej jest on zapisany jako macierz numpy, a nie w jako standardowa sekwencja Pythonowa (lista, krotka) – co oczywiście przekłada się na pewne różnice w operowaniu na tym obiekcie.

- biblioteka opencv

```
img_cv2 = cv2.imread('..../obrazy_testowe/lena_512x512.png')
plt.imshow(img_cv2[:, :, ::-1])
plt.show()
```



### Tak na prawdę wszystkie te biblioteki realizują to samo.

Dają nam przy tym wolność wyboru.

Aby z obiektu PIL.Image uzyskać macierz w postaci SZER\*\*\*x\*\*\*WYS\*\*\*x\*\*\*RGB, należy przekształcić go do obiektu np.array

```
img_np = np.array(img_pil)
print('Pillow i matplotlib.pyplot robi to samo' if np.all(img_np == np.uint8(img_plt*255)) else 'robią coś innego')
```

Pillow i matplotlib.pyplot robi to samo

*Obiekty PIL.Image umożliwiają też bardzo wiele wbudowanych transformacji wydajnymi metodami*

Jedyna różnica to, że pillow wczytuje obrazy RGB 8-bitowe (*uint8*) w skali 0-255, a matplotlib.pyplot konwertuje je na *floating* 0-1. Dlatego w celu porównania obu macierzy, zmienną *img\_plt* przeskalowano do odpowiedniego zakresu i zmieniono jej typ.

## **Obiekty PIL.Image umożliwiają też bardzo wiele wbudowanych transformacji wydajnymi metodami**

- zmiana rozmiaru obrazu

```
img_pil.resize((300,100))
```



- rotacja obrazu o zadany kąt

```
img_pil.rotate(11).resize((200,200))
```



- wybór fragmentu obrazu (o kształcie prostokąta)

```
img_pil.crop((11,11,300,300)).resize((200,200))
```



Opencv też ma wiele ciekawych funkcjonalności

A część z nich poznasz w kolejnych rozdziałach, bo głównie na tej bibliotece oparty jest ten poradnik.

## Zapis obrazów w omawianych bibliotekach

```
img_pil.save('lena.png')
cv2.imwrite('lena.png',img_cv2)
True
plt.imsave('lena.png',img_plt)
```

### 3 Implementacja operacji konwolucji w Pythonie

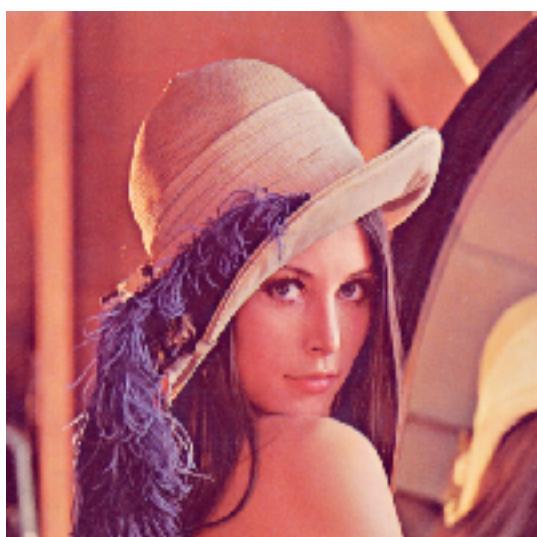
Czasem zdarza się, że chcemy sami stworzyć jakiś filtr by analizować obraz – na przykład podczas wykorzystywania obrazów w uczeniu maszynowym. Wtedy warto wiedzieć, czym jest konwolucja i jak można ją zaimplementować w łatwy sposób. Poniżej pokażemy Ci przykładowe wywołania.

A czym jest **konwolucja**? Najprościej mówiąc, jest to złożenie dwóch funkcji, w nową, trzecią funkcję. Matematycznych podstaw nie będziemy tu prezentować, polecamy za to gorąco artykuł pod tym adresem: <https://ksopyla.com/python/operacja-splotu-przetwarzanie-obrazow/>. W przetwarzaniu obrazów cyfrowych, konwolucję można rozumieć jako używanie wartości z jednej funkcji, jako maski filtrującej, która przesuwa się następnie po całym obrazie.

Tym razem będziemy potrzebować większego zestawu bibliotek – jeśli nie pamiętasz jak się je instaluje, wróć do wstępnu tego poradnika.

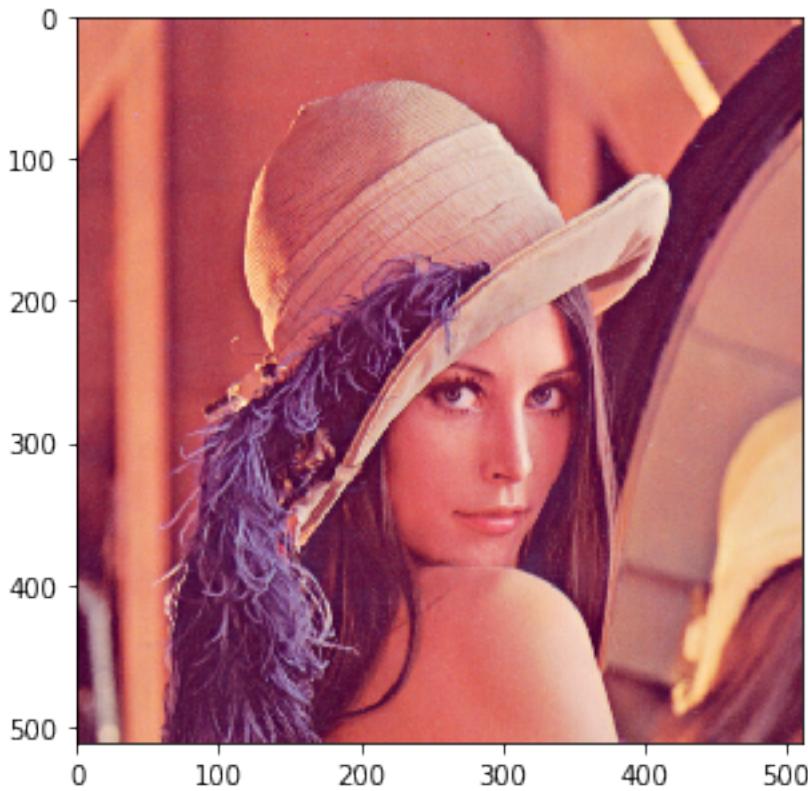
```
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import numba as nb
from numba import cuda
from scipy.ndimage.filters import convolve
import cv2

img_pil = Image.open('..../obrazy_testowe/lena_512x512.png')
img_pil.resize((200,200))
```



```
img_np = np.array(img_pil)/255
```

```
plt.figure(figsize=(5,5))
plt.imshow(img_np)
plt.show()
img_np.shape
```



(512, 512, 3)

## Filtr średniej 3x3

Pokażemy Ci działanie bardzo prostej maski, która jest po prostu średnią wartością z maski 3x3. Pamiętaj, że przy bardziej skomplikowanych przekształceniach, czas wykonywania obliczeń także się wydłuży, dlatego warto wiedzieć, jak zaoszczędzić sobie trochę czasu ;)

```
# tworzymy maskę
mean_3_3 = np.array([[1, 1, 1],
                     [1, 1, 1],
                     [1, 1, 1]],)
dtype='float')/9
```

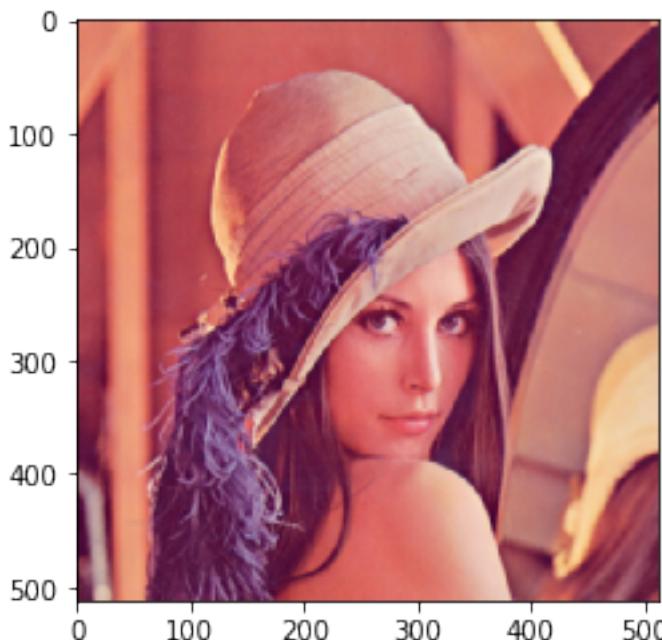
- funkcja wbudowana w bibliotekę *scipy*

```
%%timeit
img_mean = np.stack([convolve(np.squeeze(img_np[...,i]),mean_3_3) for i in range(3)],axis=-1)
```

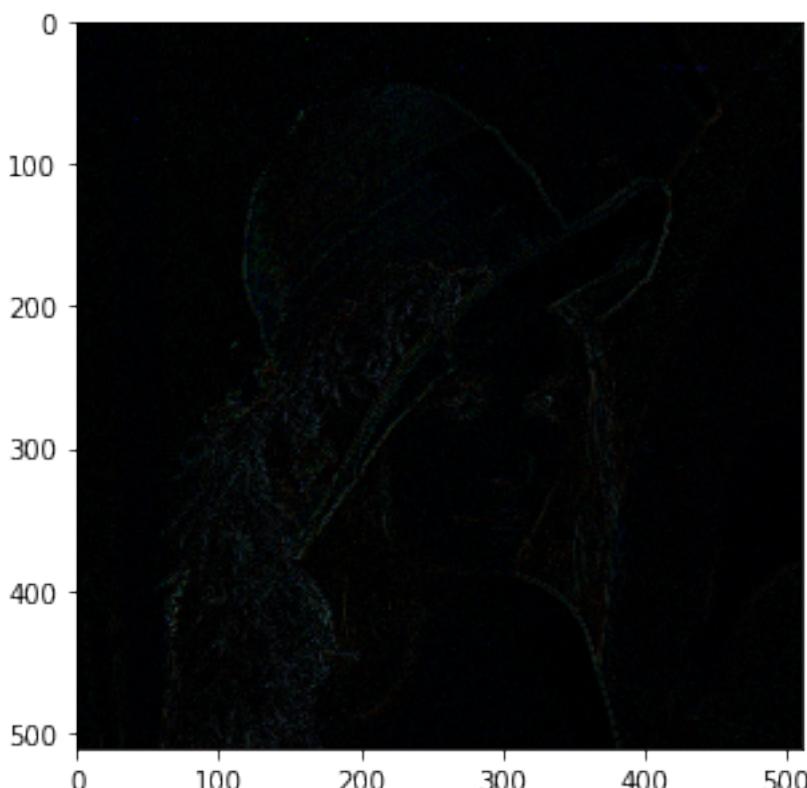
10.2 ms ± 629 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

```
img_mean = np.stack([convolve(np.squeeze(img_np[...,i]),mean_3_3) for i in range(3)],axis=-1)
plt.imshow(img_mean)
plt.show()
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer



```
plt.figure(figsize=(5,5))
plt.imshow(np.abs(img_mean-img_np))
plt.show()
```



- tym razem funkcja z *OpenCV*

```
%timeit cv2.filter2D(img_np,-1,mean_3_3)
2.6 ms ± 165 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

- jak zrobić to własnoręcznie z użyciem pętli

```
%%timeit
img_mean_fl = img_np.copy()
for i in range(1,img_mean_fl.shape[0]-1):
    for j in range(1,img_mean_fl.shape[1]-1):
        for k in range(3):
```

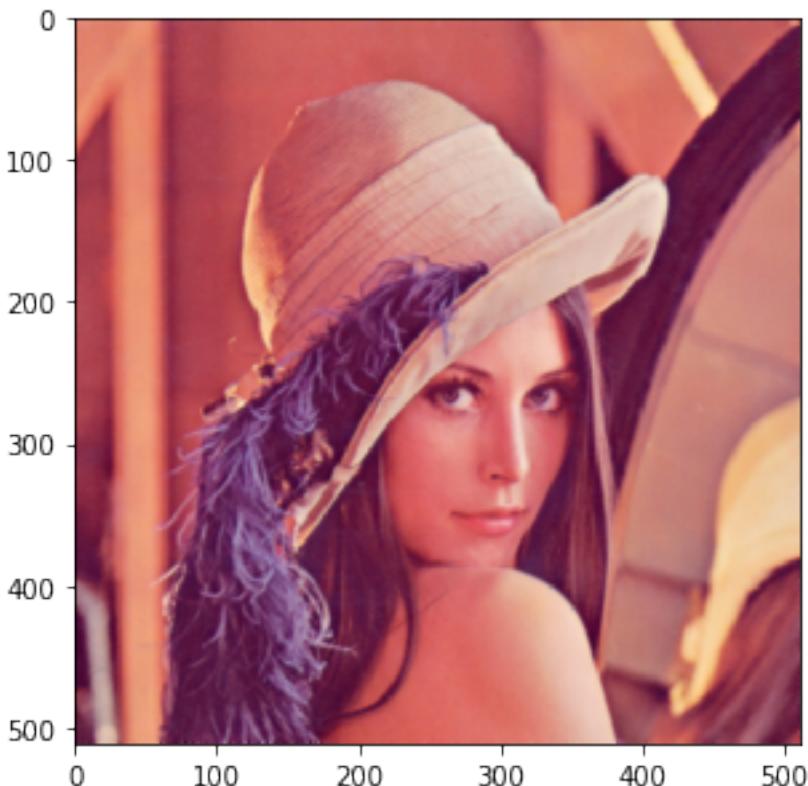
### 3 Implementacja operacji konwolucji w Pythonie

```
val = 0
for m in range(-1,2):
    for n in range(-1,2):
        val+= img_mean_fl[i+m,j+n,k]*mean_3_3[m+1,n+1]
img_mean_fl[i,j,k] = val
```

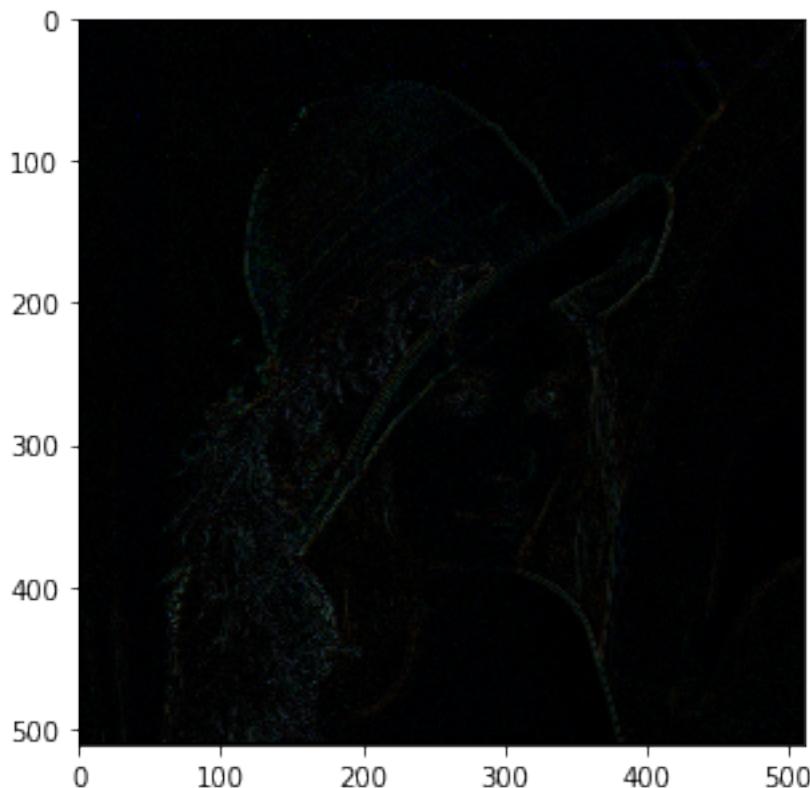
3.51 s ± 227 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

```
img_mean_fl = img_np.copy()
for i in range(1,img_mean_fl.shape[0]-1):
    for j in range(1,img_mean_fl.shape[1]-1):
        for k in range(3):
            val = 0
            for m in range(-1,2):
                for n in range(-1,2):
                    val+= img_mean_fl[i+m,j+n,k]*mean_3_3[m+1,n+1]
            img_mean_fl[i,j,k] = val
```

```
plt.figure(figsize=(5,5))
plt.imshow(img_mean_fl)
plt.show()
```



```
plt.figure(figsize=(5,5))
plt.imshow(np.abs(img_mean_fl-img_np))
plt.show()
```



- i jak to wygląda po przyspieszeniu z użyciem pakietu *numba*

```
img_mean_fl = img_np.copy()
@nb.jit()
def wolna_funkcja_duzo_petli(img_mean_fl):
    for i in range(1,img_mean_fl.shape[0]-1):
        for j in range(1,img_mean_fl.shape[1]-1):
            for k in range(3):
                val = 0
                for m in range(-1,2):
                    for n in range(-1,2):
                        val+= img_mean_fl[i+m,j+n,k]*mean_3_3[m+1,n+1]
                img_mean_fl[i,j,k] = val
    return img_mean_fl

%time img_mean_fl = wolna_funkcja_duzo_petli(img_mean_fl)
CPU times: user 160 ms, sys: 36.9 ms, total: 197 ms
Wall time: 146 ms
```



## 4 Filtracja

Jednym z ważnych zagadnień w dziedzinie przetwarzania obrazów jest proces filtracji. Dzięki niemu można przygotować obrazy do dalszej obróbki lub wydobyć interesujące nas informacje. Filtracja jest niczym innym jak pewną operacją matematyczną na każdym pikselu, w oparciu o informacje zebrane z wielu innych pikseli w obrazie. Choć czasem wzory matematyczne wydają się skomplikowane, na szczęście nie musimy sami implementować tych algorytmów, gdyż w znakomitej większości są one już gotowe. Trzeba tylko wiedzieć, gdzie szukać.

Poniżej prezentujemy jak można przeprowadzić filtrację uśredniającą, gaussowską oraz medianową. Najpierw na wybrany obraz dodaliśmy szum gaussowski oraz szum typu ‘sól i pieprz’ (*salt&pepper*), by w kolejnym kroku ją odfiltrować, co pokaże nam skuteczność tej operacji.

Standardowo zaczynamy od wczytania potrzebnych bibliotek oraz otwarcia obrazu (jeśli nie rozumiesz któregoś z zapisów, wróć do poprzedniego rozdziału).

```
import cv2
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

img = Image.open('../obrazy_testowe/lena_512x512.png')
imgColArray = np.array(img)/255
```

Teraz dodamy szum do obrazu, a wynik zapiszemy jako nową zmienną, by móc się jeszcze do niej odwołać później.

```
# szum Gaussa
row, col, ch= imgColArray.shape
mean = 0
var = 0.1
sigma = var**0.8
gauss = np.random.normal(mean, sigma, (row,col,ch))
gauss = gauss.reshape(row, col, ch)
imgGauss = imgColArray + gauss

# szum sól i pieprz
sp = 0.5
amount = 0.05
imgSaltPepper = np.copy(imgColArray)

numSalt = np.ceil(amount * imgColArray.size * sp)
coords = [np.random.randint(0, i - 1, int(numSalt)) for i in imgColArray.shape]
imgSaltPepper[tuple(coords)] = 1

numPepper = np.ceil(amount* imgColArray.size * (1. - sp))
coords = [np.random.randint(0, i - 1, int(numPepper)) for i in imgColArray.shape]
imgSaltPepper[tuple(coords)] = 0
```

Na tym etapie pozostało nam jeszcze jedynie wyświetlić wszystkie trzy obrazy, by sprawdzić, czy uzyskaliśmy żądzony efekt.

```
plt.figure(figsize = (15, 10))
```

```
plt.subplot(131)
plt.imshow(imgColArray)
plt.title('Obraz oryginalny')
plt.axis('off')

plt.subplot(132)
plt.imshow(imgGauss)
```

#### 4 Filtracja

```
plt.title('Obraz z szumem Gaussa')
plt.axis('off')

plt.subplot(133)
plt.imshow(imgSaltPepper)
plt.title('Obraz z szumem "sól i pieprz"')
plt.axis('off')

plt.show()

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)
```



Tak uzyskane obrazy poddamy procesowi filtracji. Zauważ, że każdą filtrację przeprowadzimy dla trzech wielkości masek, co od razu pokaże wpływ rozmiaru maski na wynik.

### Filtracja uśredniająca

Filtrację uśredniającą realizują dwie funkcje zaimplementowane w bibliotece Open CV (*cv2*): *blur()* oraz *boxFilter()*. Dokładny ich opis oraz nieznaczne różnice można znaleźć w dokumentacji - my użyjemy funkcji *cv2.blur()*. W najprostszym zastosowaniu należy podać jako argumenty obraz, który poddamy filtracji oraz wielkość maski (szerokość i wysokość). Dodatkowo można też określić punkt zaczepienia (domyślnie jest to środek maski), a także sposób ekstrapolacji pikseli znajdujących się poza krawędziami obrazu.

```
imgGnoiseFavg3 = cv2.blur(imgGauss, (3,3))
imgGnoiseFavg9 = cv2.blur(imgGauss, (9,9))
imgGnoiseFavg15 = cv2.blur(imgGauss, (15,15))

imgSnoiseFavg3 = cv2.blur(imgSaltPepper, (3,3))
imgSnoiseFavg9 = cv2.blur(imgSaltPepper, (9,9))
imgSnoiseFavg15 = cv2.blur(imgSaltPepper, (15,15))

plt.figure(figsize = (15, 10))

plt.subplot(231)
plt.imshow(imgGnoiseFavg3)
plt.title('Szum Gaussa, maska 3x3')
plt.axis('off')
plt.subplot(232)
plt.imshow(imgGnoiseFavg9)
plt.title('Szum Gaussa, maska 9x9')
plt.axis('off')
plt.subplot(233)
plt.imshow(imgGnoiseFavg15)
plt.title('Szum Gaussa, maska 15x15')
plt.axis('off')

plt.subplot(234)
plt.imshow(imgSnoiseFavg3)
plt.title('Szum "sól i pieprz", maska 3x3')
```

```

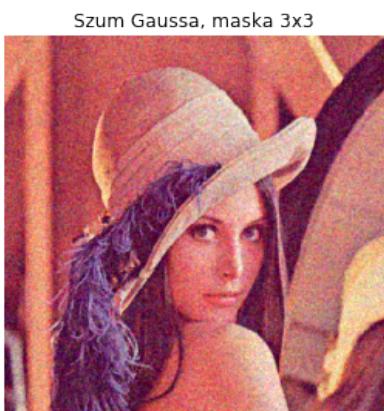
plt.axis('off')
plt.subplot(235)
plt.imshow(imgSnoiseFavg9)
plt.title('Szum "sól i pieprz", maska 9x9')
plt.axis('off')
plt.subplot(236)
plt.imshow(imgSnoiseFavg15)
plt.title('Szum "sól i pieprz", maska 15x15')
plt.axis('off')

plt.suptitle("Filtracja uśredniająca", fontsize=16)
plt.show()

```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)  
 Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)  
 Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers)

Filtracja uśredniająca



## Filtracja gaussowska

Efekt rozmycia obrazu możemy uzyskać również wykorzystując filtr Gaussa – użyjemy w tym celu funkcji *GaussianBlur()*. Tutaj również należy podać obraz wejściowy, wielkość maski, a także można określić jak ekstrapolować piksele “rozszerzające” obraz. 0 jest parametrem *sigma*, który określa wielkość odchylenia standardowego (jedna ze zmiennych we wzorze matematycznym). Określa się go zarówno dla kierunku X jak i Y. Można pominąć ten parametr (tak jak my w kierunku Y), ale wtedy zostanie on obliczony z odpowiedniego przekształcenia wielkości maski – jeśli chcesz mieć pełną kontrolę nad działaniem tej funkcji, lepiej zdefiniować wszystkie parametry.

```

imgGnoiseFgauss3 = cv2.GaussianBlur(imgGauss, (3,3), 0)
imgGnoiseFgauss9 = cv2.GaussianBlur(imgGauss, (9,9), 0)
imgGnoiseFgauss15 = cv2.GaussianBlur(imgGauss, (15,15), 0)

imgSnoiseFgauss3 = cv2.GaussianBlur(imgSaltPepper, (3,3), 0)

```

#### 4 Filtracja

```
imgSnoiseFgauss9 = cv2.GaussianBlur(imgSaltPepper, (9,9), 0)
imgSnoiseFgauss15 = cv2.GaussianBlur(imgSaltPepper, (15,15), 0)

plt.figure(figsize = (15, 10))

plt.subplot(231)
plt.imshow(imgGnoiseFgauss3)
plt.title('Szum Gaussa, maska 3x3')
plt.axis('off')
plt.subplot(232)
plt.imshow(imgGnoiseFgauss9)
plt.title('Szum Gaussa, maska 9x9')
plt.axis('off')
plt.subplot(233)
plt.imshow(imgGnoiseFgauss15)
plt.title('Szum Gaussa, maska 15x15')
plt.axis('off')

plt.subplot(234)
plt.imshow(imgSnoiseFgauss3)
plt.title('Szum "sól i pieprz", maska 3x3')
plt.axis('off')
plt.subplot(235)
plt.imshow(imgSnoiseFgauss9)
plt.title('Szum "sól i pieprz", maska 9x9')
plt.axis('off')
plt.subplot(236)
plt.imshow(imgSnoiseFgauss15)
plt.title('Szum "sól i pieprz", maska 15x15')
plt.axis('off')

plt.suptitle("Filtracja Gaussowska", fontsize=16)
plt.show()
```

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer

Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer

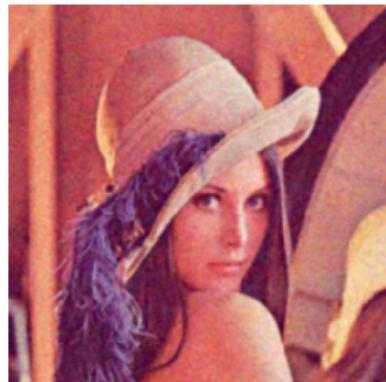
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer

## Filtracja Gaussowska

Szum Gaussa, maska 3x3



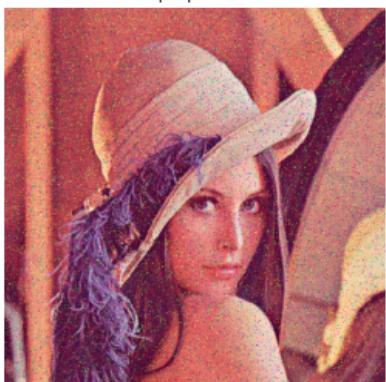
Szum Gaussa, maska 9x9



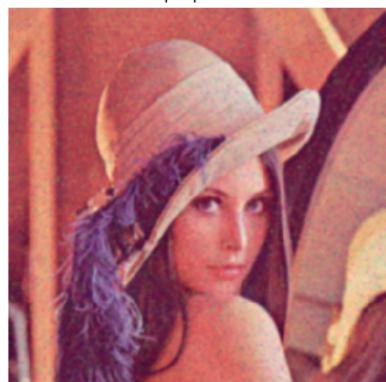
Szum Gaussa, maska 15x15



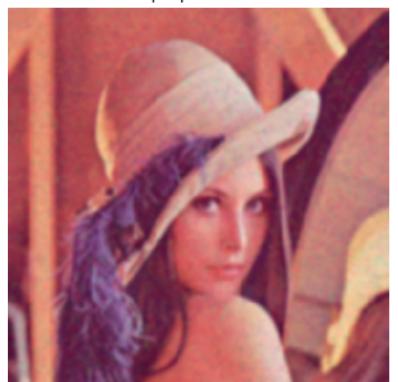
Szum "sól i pieprz", maska 3x3



Szum "sól i pieprz", maska 9x9



Szum "sól i pieprz", maska 15x15



## Filtracja medianowa

Filtrację medianową zrealizujemy za pomocą funkcji `medianBlur()`. Wymaga ona podania obrazu w formacie `uint8`, dlatego musimy przeskalać nasz obraz. W samej funkcji poza podaniem obrazu wejściowego, definiujemy tylko rozmiar maski (okna przetwarzania), który musi być liczbą całkowitą większą od 1.

```
imgGauss = np.array(imgGauss*255, dtype = np.uint8)
imgSaltPepper = np.array(imgSaltPepper*255, dtype = np.uint8)

imgGnoiseFmedian3 = cv2.medianBlur(imgGauss, 3)
imgGnoiseFmedian9 = cv2.medianBlur(imgGauss, 9)
imgGnoiseFmedian15 = cv2.medianBlur(imgGauss, 15)

imgSnoiseFmedian3 = cv2.medianBlur(imgSaltPepper, 3)
imgSnoiseFmedian9 = cv2.medianBlur(imgSaltPepper, 9)
imgSnoiseFmedian15 = cv2.medianBlur(imgSaltPepper, 15)

plt.figure(figsize = (15, 10))

plt.subplot(231)
plt.imshow(imgGnoiseFmedian3)
plt.title('Szum Gaussa, maska 3x3')
plt.axis('off')
plt.subplot(232)
plt.imshow(imgGnoiseFmedian9)
plt.title('Szum Gaussa, maska 9x9')
plt.axis('off')
plt.subplot(233)
plt.imshow(imgGnoiseFmedian15)
plt.title('Szum Gaussa, maska 15x15')
```

#### 4 Filtracja

```
plt.axis('off')
plt.subplot(234)
plt.imshow(imgSnoiseFmedian3)
plt.title('Szum "sól i pieprz", maska 3x3')
plt.axis('off')
plt.subplot(235)
plt.imshow(imgSnoiseFmedian9)
plt.title('Szum "sól i pieprz", maska 9x9')
plt.axis('off')
plt.subplot(236)
plt.imshow(imgSnoiseFmedian15)
plt.title('Szum "sól i pieprz", maska 15x15')
plt.axis('off')

plt.suptitle("Filtracja medianowa", fontsize=16)
plt.show()
```

Filtracja medianowa

Szum Gaussa, maska 3x3



Szum Gaussa, maska 9x9



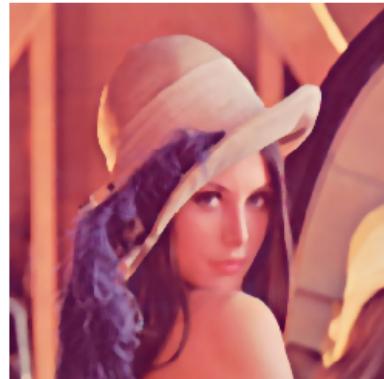
Szum Gaussa, maska 15x15



Szum "sól i pieprz", maska 3x3



Szum "sól i pieprz", maska 9x9



Szum "sól i pieprz", maska 15x15



Jak widzisz, do różnych typów zakłóceń warto wykorzystać inne metody filtracji. Niestety nie istnieje jakieś "idealne rozwiązanie", które będzie w stanie poprawić nam każdy obraz, dlatego należy wiedzieć, jakie efekty osiągane są przez daną metodę filtracji. Zauważ też, że przy stosowaniu większych masek, stracimy również więcej informacji.

# 5 Operacje morfologiczne

Przekształcenia morfologiczne cyfrowych obrazów, to takie przekształcenia, w wyniku których struktura lub forma obiektu na obrazie zostaje zmieniona. Dokonujemy ich na obrazie binarnym, czyli czarno-białym, a operacje morfologiczne polegają po prostu na zamianie wartości odpowiednich pikseli na przeciwnie. Podstawowe przekształcenia to dylatacja, erozja i szkieletyzacja, które można ze sobą łączyć, co daje podstawę do budowania skomplikowanych systemów analizy obrazu.

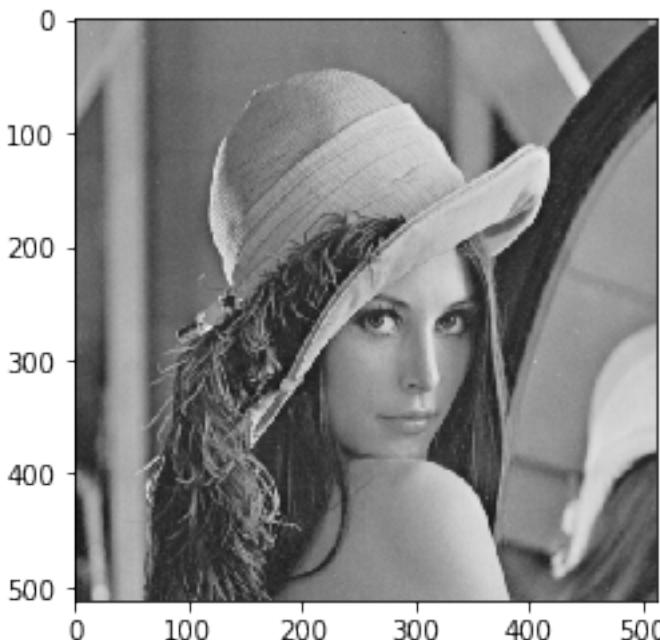
```
from PIL import Image
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

## Wczytanie obrazu

W celu wykonania operacji morfologicznych na obrazie, należy najpierw go wczytać i przekonwertować do postaci monochromatycznej (czyli do skali szarości). Oczywiście można tego dokonać na kilka sposobów, np. za pomocą komendy *Image.open()* z biblioteki Pillow oraz metody *convert* z parametrem 'L'.

Innym sposobem będzie ustawienie odpowiedniej metody w funkcji *imread()* z pakietu OpenCV: *cv2.imread(obraz, cv2.IMREAD\_GRAYSCALE)*

```
img = Image.open('..../obrazy_testowe/lena_512x512.png').convert('L')
plt.imshow(img)
plt.show()
```



## Binaryzacja

Następnie obraz należy zamienić na postać macierzową, za pomocą metody *asarray()* z biblioteki *numpy*. Na tak przetworzonym obrazie, można dokonać operacji binaryzacji, która polega na przeskalowania obrazu do dwóch wartości, w zależności od przyjętego progu. Najczęściej nowe wartości pikseli zapisuje się jako 0 i 1, ale możesz spotkać się też z innymi parami wartości, np: (0, 255), (-1, 1), (*True*, *False*). Poniżej przedstawiono przykład binaryzacji dla trzech progów odcięcia – 25%, 50% i 75% maksymalnej wartości (u nas to 255). Operacji binaryzacji dokonano za pomocą metody *threshold()* z biblioteki *OpenCV*. Do wyświetlenia obrazów użyto tym razem metody *imshow()* z pakietu *matplotlib.pyplot*, ponieważ zostały one wcześniej przekonwertowane na postać macierzową.

## 5 Operacje morfologiczne

```
imgArray = np.asarray(img)
ret1, imgThresholded1 = cv2.threshold(imgArray, 64, 255, cv2.THRESH_BINARY)
ret2, imgThresholded2 = cv2.threshold(imgArray, 127, 255, cv2.THRESH_BINARY)
ret3, imgThresholded3 = cv2.threshold(imgArray, 191, 255, cv2.THRESH_BINARY)

plt.figure(1, figsize = (15, 10))

plt.subplot(131)
plt.title('Próg 64')
plt.axis('off')
plt.imshow(imgThresholded1, cmap = 'gray')

plt.subplot(132)
plt.title('Próg 127')
plt.axis('off')
plt.imshow(imgThresholded2, cmap = 'gray')

plt.subplot(133)
plt.title('Próg 191')
plt.axis('off')
plt.imshow(imgThresholded3, cmap = 'gray')

plt.show()
```



Jak widzisz, próg odcięcia diametralnie może zmienić obraz wynikowy, dlatego jego odpowiednie dobranie jest niezwykle ważne. Czasem robi się to ręcznie (najczęściej w oparciu o histogram obrazu), jednak istnieją także automatyczne metody i do dalszych operacji wykorzystano jedną z nich – metodę Otsu. Zauważ, że zmienia się teraz nieco sposób zapisu parametrów: próg odcięcia podajemy jako równy 0, a metodę Otsu wywołujemy poprzez dodanie `cv2.THRESH_OTSU`

```
retOtsu, imgThresholdedOtsu = cv2.threshold(imgArray, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)

plt.figure(1, figsize = (5, 5))
plt.imshow(imgThresholdedOtsu, cmap = 'gray')
plt.title('Metoda Otsu')
plt.axis('off')
plt.show()
```

Metoda Otsu



## Operacje morfologiczne

Na tak przetworzonym obrazie, można już wykonywać operacje morfologiczne. Konieczny jest wybór elementu strukturalnego (kernel, maska) oraz jego wielkości. W poniższym przykładzie wykorzystano kwadrat o wielkości 5 pikseli, ale może to być w zasadzie element o dowolnym kształcie (oczywiście o ile jest mniejszy od obrazu ;)). Poprzez nakładanie takiej maski, można wykonać szereg operacji, jak na przykład erozję i dylatację czy otwarcie i zamknięcie (które są odpowiednią kombinacją erozji oraz dylatacji). W każdym przypadku element strukturalny przesuwa się wzduł obrazu, a następnie dla każdego piksela sprawdza jego sąsiedztwo. W zależności od wyboru algorytmu, można zaobserwować następujące różnice na obrazach wynikowych.

```
kernel = np.ones((5,5), np.uint8)
imgErosion = cv2.erode(imgThresholdedOtsu, kernel, iterations = 1)
imgDilation = cv2.dilate(imgThresholdedOtsu, kernel, iterations = 1)
```

Jak widzisz podstawowe operacje morfologiczne doczekały się nawet własnych funkcji. Niżej użyliśmy jeszcze funkcji *morphologyEx()*, która posiada wiele innych metod, będących rozwinięciem tych przekształceń.

```
imgOpening = cv2.morphologyEx(imgThresholdedOtsu, cv2.MORPH_OPEN, kernel)
imgClosing = cv2.morphologyEx(imgThresholdedOtsu, cv2.MORPH_CLOSE, kernel)
imgGradient = cv2.morphologyEx(imgThresholdedOtsu, cv2.MORPH_GRADIENT, kernel)
imgBlackhat = cv2.morphologyEx(imgThresholdedOtsu, cv2.MORPH_BLACKHAT, kernel)
```

```
plt.figure(1, figsize = (15, 10))

plt.subplot(231)
plt.imshow(imgErosion, cmap = 'gray')
plt.title('Erozja')
plt.axis('off')

plt.subplot(232)
plt.imshow(imgDilation, cmap = 'gray')
plt.title('Dylatacja')
plt.axis('off')

plt.subplot(233)
plt.imshow(imgOpening, cmap = 'gray')
```

## 5 Operacje morfologiczne

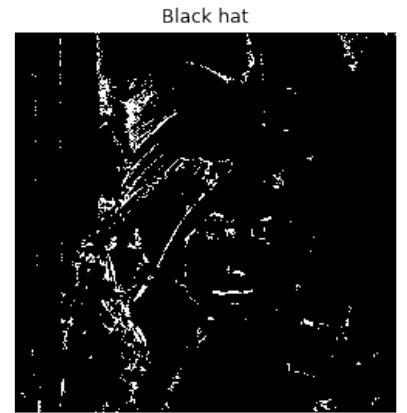
```
plt.title('Otwarcie')
plt.axis('off')

plt.subplot(234)
plt.imshow(imgClosing, cmap = 'gray')
plt.title('Zamknięcie')
plt.axis('off')

plt.subplot(235)
plt.imshow(imgGradient, cmap = 'gray')
plt.title('Gradient')
plt.axis('off')

plt.subplot(236)
plt.imshow(imgBlackhat, cmap = 'gray')
plt.title('Black hat')
plt.axis('off')

plt.show()
```



# 6 Detekcja obiektów

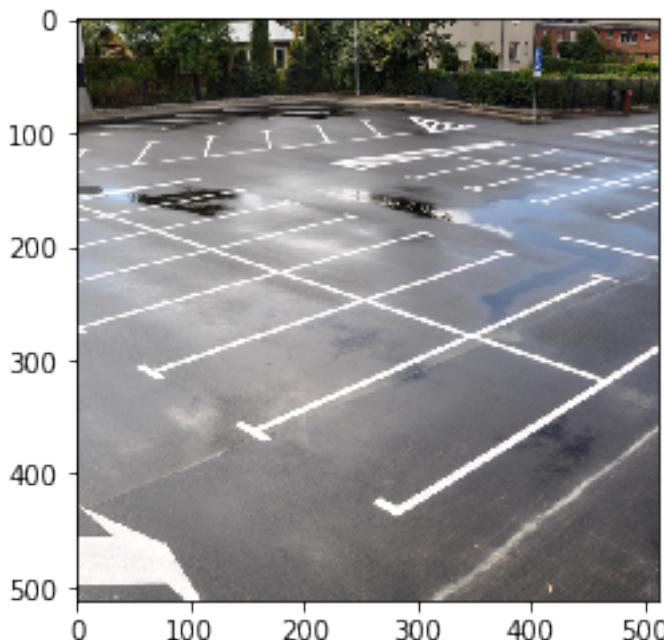
Kolejnym zagadnieniem istotnym w analizie obrazów, jest detekcja obiektów. Może ona być pomocna w szacowaniu ilości pewnego typu obiektów na obrazie czy też wyznaczania ich dokładnego położenia. Tutaj pokażemy kilka typowych przekształceń opierających się na wyszukiwaniu określonych kształtów.

```
import cv2
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
```

## Detekcja linii

Po załadowaniu niezbędnych bibliotek, wczytano obraz na dwa znanego Ci już sposoby: za pomocą *Image.open()* (po to, by go wyświetlić w poradniku) oraz za pomocą *cv2.imread()* – ten będzie wejściem do kolejnych funkcji.

```
img = Image.open('../obrazy_testowe/parking_512x512.png')
imgColArray = cv2.imread('../obrazy_testowe/parking_512x512.png')
imgGrey = img.convert('L')
imgArray = np.asarray(imgGrey)
plt.figure(1, figsize = (7, 7))
plt.imshow(img)
plt.show()
```



Podobnie jak przy operacjach morfologicznych, tutaj też pracujemy na obrazach binarnych. Tym razem dobraślimy ręcznie próg odcięcia, by lepiej wyodrębnić interesujące nas obiekty - linie. Na parkingu są one białe, dlatego ustalony próg jest dość wysoki (wartość 200).

```
ret, imgBin = cv2.threshold(imgArray, 200, 255, cv2.THRESH_BINARY)

plt.figure(1, figsize = (7, 7))
plt.imshow(imgBin, cmap = 'gray')
plt.title('Obraz zbinaryzowany')
plt.axis('off')
plt.show()
```



Detekcji linii dokonaliśmy za pomocą funkcji *cv2.HoughLinesP()* (możesz także użyć mniej zoptymalizowanej funkcji *cv2.HoughLines()*, ale obliczenia zajmują zdecydowanie więcej czasu). W niej należy zdefiniować:

- obraz wejściowy (binarny),
- dwa parametry opisujące linię w układzie współrzędnych biegunkowych:
  - rho: rozdzielcość podana w pikselach (u nas równa 1),
  - theta: rozdzielcość podana w radianach – u nas równa 1 stopień, czyli  $\pi/180$ ,
- minimalna liczba przecięć “wyszukujących” linii – jego zmiana znacząco wpływa na wynik,
- minimalna długość poszukiwanej linii (podana w pikselach),
- maksymalna przerwa w obrębie jednej linii (także w pikselach).

Znalezione linie wyrysowaliśmy za pomocą funkcji *cv2.line()*, na podstawie współrzędnych linii odnalezionych przez detektor. Funkcja *np.size()* z parametrem *0* zlicza wiersze macierzy, które u nas są liczbą znalezionych linii.

```
minLineLength = 10
maxLineGap = 10
lines = cv2.HoughLinesP(imgBin, 1, np.pi/180, 150, minLineLength, maxLineGap)
for line in lines:
    for x1,y1,x2,y2 in line:
        cv2.line(imgColArray, (x1,y1), (x2,y2), (255,0,0), 4)

countLines = np.size(lines, 0)
plt.figure(1, figsize = (7, 7))
plt.imshow(imgColArray)
plt.title(str(countLines) + ' znalezione linie')
plt.axis('off')
plt.show()
```

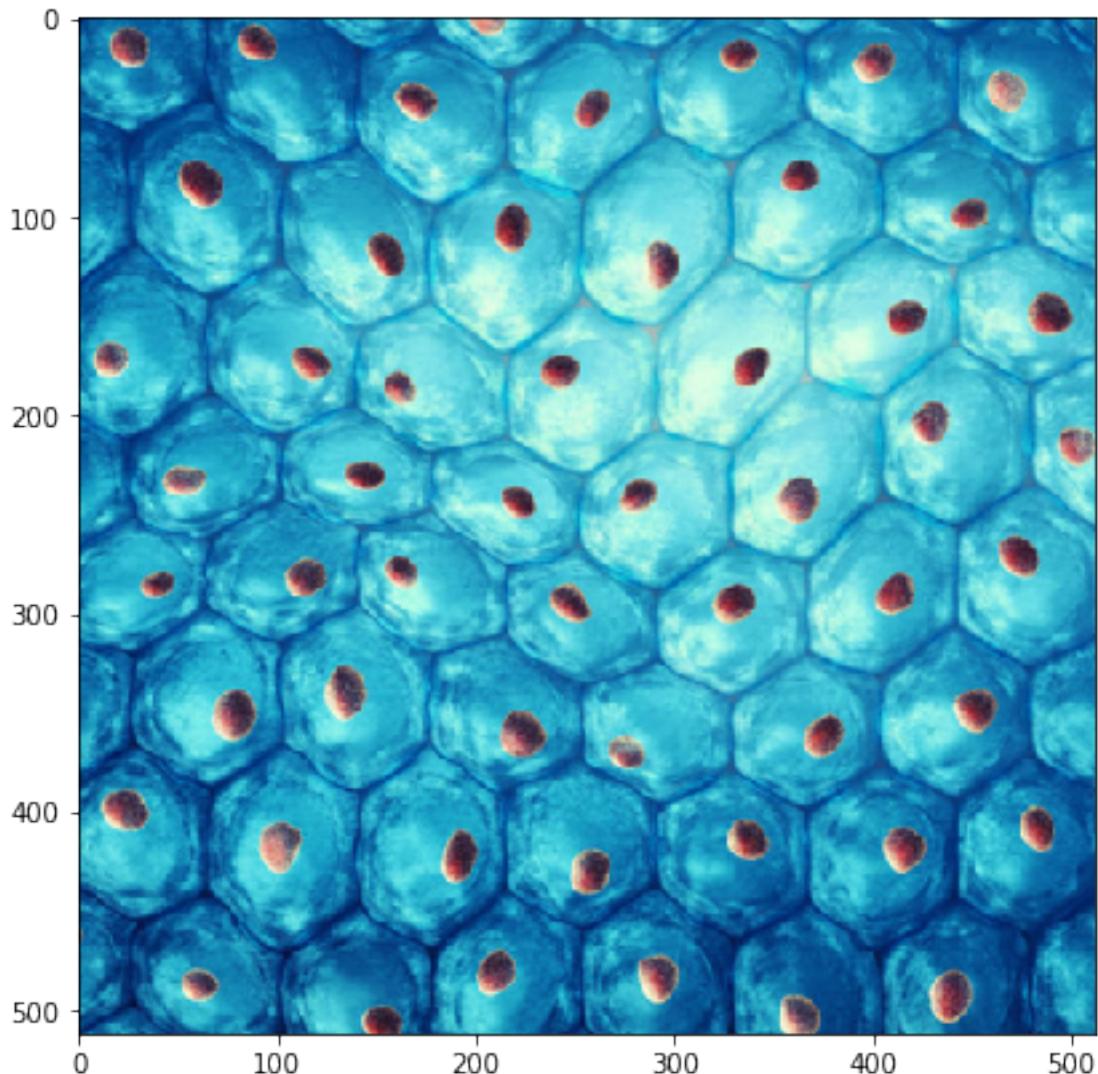
44 znalezione linie



## Detekcja obiektów niejednorodnych

Innym przykładem zastosowania metod detekcji, może być poszukiwanie obiektów, które nie posiadają zdefiniowanego kształtu, tak jak poniższe komórki i ich jądra komórkowe.

```
img = Image.open('../obrazy_testowe/cells_512x512.png')
imgGrayArray = cv2.imread('../obrazy_testowe/cells_512x512.png', cv2.IMREAD_GRAYSCALE)
plt.figure(1, figsize = (7, 7))
plt.imshow(img)
plt.show()
```



Tym razem użyliśmy funkcji `cv2.SimpleBlobDetector()`, dla której najpierw zdefiniowaliśmy takie parametry jak: minimalna i maksymalna szukana powierzchnia obiektów, współczynniki okrągłości, stałości czy wydłużenia obiektów, a także próg odcięcia. Następnie dokonano przeszukania obrazu (`detector.detect()`), z uwzględnieniem zdefiniowanych parametrów. Znalezione obiekty zaznaczono okręgami za pomocą funkcji `cv2.drawKeypoints()`. Przyjmuje ona:

- obraz wejściowy,
- macierz z punktami, w których znajdują się znalezione obiekty,
- pustą macierz na obraz wynikowy,
- opis koloru zaznaczenia w przestrzeni RGB (u nas będzie to czerwony),
- dalsze flagi, których dokładne opisy znajdziesz w dokumentacji.

```
help(cv2.drawKeypoints)
```

Help on built-in function drawKeypoints:

```
drawKeypoints(...)
    drawKeypoints(image, keypoints, outImage[, color[, flags]]) -> outImage
    .   @brief Draws keypoints.
    .
    .   @param image Source image.
    .   @param keypoints Keypoints from the source image.
    .   @param outImage Output image. Its content depends on the flags value defining what is drawn in the
    .   output image. See possible flags bit values below.
    .   @param color Color of keypoints.
    .   @param flags Flags setting drawing features. Possible flags bit values are defined by
    .   DrawMatchesFlags. See details above in drawMatches .
    .
    .   @note
    .   For Python API, flags are modified as cv2.DRAW_MATCHES_FLAGS_DEFAULT,
```

```

. cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS, cv2.DRAW_MATCHES_FLAGS_DRAW_OVER_OUTIMG,
. cv2.DRAW_MATCHES_FLAGS_NOT_DRAW_SINGLE_POINTS

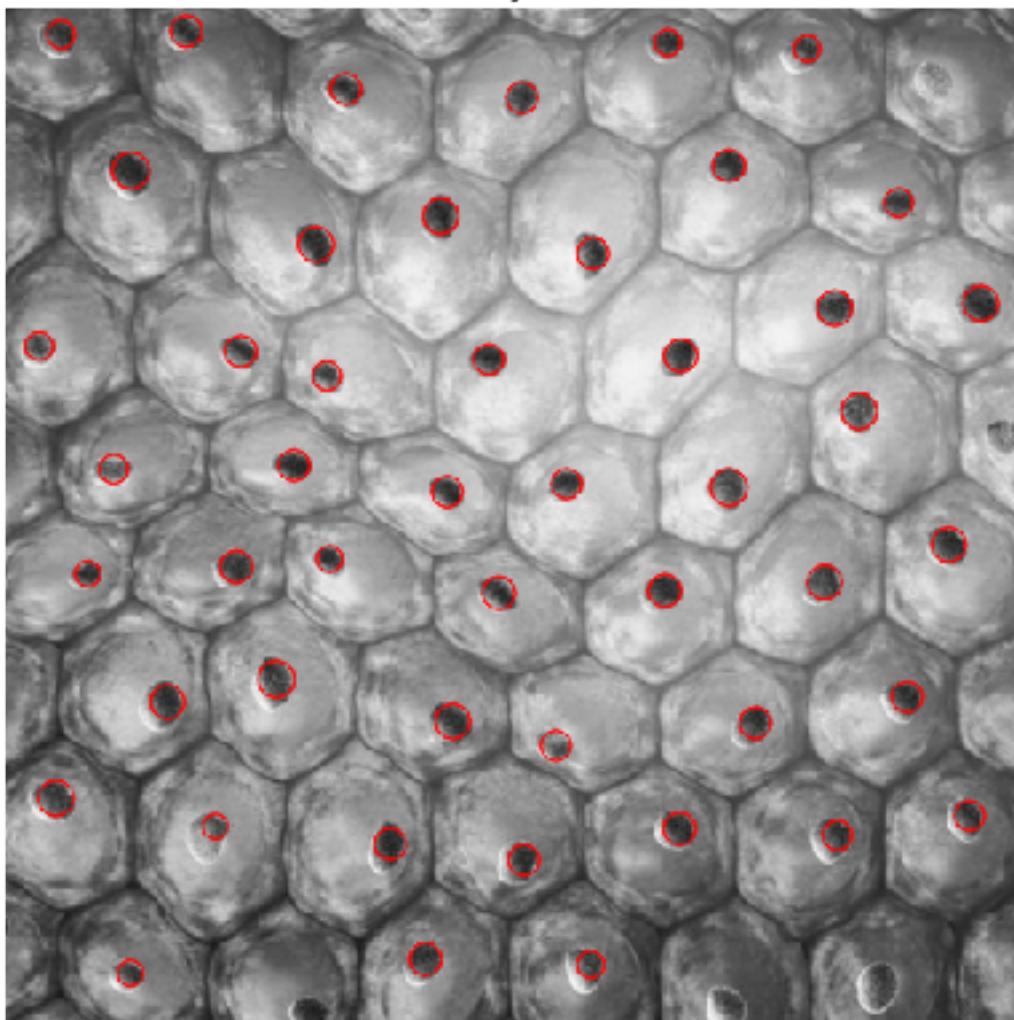
params = cv2.SimpleBlobDetector_Params()
params.filterByArea = True
params.filterByColor = False
params.filterByCircularity = True
params.minThreshold = 0
params.maxThreshold = 255
params.minArea = 100
params.maxArea = 300
params.minCircularity = 0.52
params.minConvexity = 0.1
params.minInertiaRatio = 0.1

detector = cv2.SimpleBlobDetector_create(params)
objects = detector.detect(imgGrayArray)
imgWithObjects = cv2.drawKeypoints(imgGrayArray, objects, np.array([]), (255, 0, 0),
                                   cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)

countObj = len(objects)
plt.figure(1, figsize = (7, 7))
plt.imshow(imgWithObjects)
plt.title(str(countObj) + ' znalezionych obiektów')
plt.axis('off')
plt.show()

```

48 znalezionych obiektów



Zauważ, że można także filtrować obiekty uwzględniając ich kolor – uda Ci się policzyć jądra komórkowe w ten sposób?

## Detekcja okręgów

Często może również istnieć potrzeba odnalezienia okręgów na obrazie – tak jak na zdjęciu zawierającym znaki drogowe. Po wczytaniu obrazu dokonano binaryzacji, w celu lepszego wyodrębnienia okręgów.

```
img = Image.open('../obrazy_testowe/signs_512x205.png')
imgGrayArray = cv2.imread('../obrazy_testowe/signs_512x205.png', cv2.IMREAD_GRAYSCALE)
ret, imgBin = cv2.threshold(imgGrayArray, 0, 255, cv2.THRESH_BINARY+cv2.THRESH_OTSU)
plt.figure(1, figsize = (9, 9))
plt.imshow(img)
plt.show()
```



Do detekcji użyto funkcji `cv2.HoughCircles()`, w niej definiujemy:

- obraz wejściowy (zbinaryzowany),
- metodę detekcji (obecnie zaimplementowana jest tylko `cv2.HOUGH_GRADIENT`),
- współczynnik rozdzielczości akumulatora, który jest odwrotny do rozdzielczości obrazu – przy wartości 1 oba będą mieć ten sam rozmiar,
- minimalną odległość pomiędzy szukanymi okręgami (w pikselach),
- dwa parametry określające progi odcięcia do metody `cv2.HOUGH_GRADIENT`,
- minimalną i maksymalną wartość promienia wyszukiwanych okręgów (podana w pikselach).

Okręgi wyrysowano następnie za pomocą funkcji `cv2.circle()`, podając jako parametry współrzędne okręgów, a także ich kolor i grubość.

```
circles = cv2.HoughCircles(imgBin, cv2.HOUGH_GRADIENT, 1, 40, param1=50, param2=20, minRadius=5, maxRadius=50)

circles = np.around(circles)
for i in circles[0,:]:
    # wyrysowanie okręgu zewnętrznego
    cv2.circle(imgGrayArray, (i[0],i[1]), i[2], (0,0,0), 3)
    # wyrysowanie środka okręgu
    cv2.circle(imgGrayArray, (i[0],i[1]), 2, (0,0,0), 4)

countCirc = np.size(circles[0], 0)
plt.figure(1, figsize = (9, 9))
plt.imshow(imgGrayArray)
plt.title(str(countCirc) + ' znalezionych okręgów')
plt.axis('off')
plt.show()
```

## 7 znalezionych okręgów



Przykładowa macierz znalezionych okręgów może wyglądać następująco. Każdy wiersz zawiera współrzędne ( $x, y$ ) środka okręgu oraz jego promień.

```
circles  
array([[476.,  78.,  28.],  
       [ 42.,  80.,  28.],  
       [414.,  76.,  18.],  
       [182.,  78.,  18.],  
       [120.,  76.,  26.],  
       [364.,  80.,  13.],  
       [230.,  78.,  12.]], dtype=float32)
```

Przy próbie dostosowania parametrów poszczególnych detektorów można zaobserwować, iż są one bardzo wrażliwe na ich zmianę. Często zatem wyniki detekcji nie są w pełni zadowalające – brakuje niektórych obiektów lub jest ich za dużo. Powoduje to konieczność poświęcenia większej uwagi tej części detekcji.