Politechnika Świętokrzyska
Wydział Elektrotechniki, Automatyki i Informatyki
Katedra Elektrotechniki Przemysłowej i Automatyki
Zakład Urządzeń i Systemów Automatyki

Cyfrowe przetwarzanie sygnałów

Zapoznanie z narzędziami do przetwarzania i wizualizacji sygnałów

Język Python · Moduł NumPy · Biblioteka Matplotlib

Instrukcja laboratoryjna
(wersja robocza)

Robert Kazała
Paweł Strączyński
2017

Cel ćwiczenia
Celem ćwiczenia jest zapoznanie z podstawami języka Python oraz bibliotekami pozwalającymi na przetwarzanie i wizualizację sygnałów w postaci graficznej.

1. Podstawy języka Python

Język Python jest interpretowanym, obiektowym językiem wysokiego poziomu. Język cechuje rozbudowana biblioteka standardowa, dynamiczne typowanie oraz automatyczne przydzielanie pamięci. Charakterystyczny dla Pythona jest fakt, że wymaga on odpowiedniego formatowania kodu – wcięcia służą do wyróżniania poszczególnych bloków kodu.

1.1. *Deklarowanie zmiennych, podstawowe operacje na liczbach i ciągach znakowych.*

Python jest językiem dynamicznie typowanym. Utworzenie zmiennej odbywa się podczas przypisania nazwie zmiennej wartości – listing 1.1. Tworzenie stringów odbywa się z wykorzystaniem jednego lub dwóch cudzysłowów.

Listing 1.1. Tworzenie zmiennych w języku Python

```
foo = 1
bar = 'example text'
baz = ''text2''
spam = 1.3 #zmienna typu rzeczywistego
eggs = 1+2j #liczba zespolona
```

Ponadto język Python cechuję silne typowanie – każde wyrażenie ma ustalony typ. Oznacza to, że konieczna jest jawna konwersja między poszczególnymi typami – listing 1.2. Funkcje do konwersji typów przedstawiono w tabeli 1.1 (język Python posiada 4 typy numeryczne).

Tabela 1.1. Funkcje do konwersji typów

Funkcja	Opis
int()	Konwersja na liczbę całkowitą
float()	Konwersja na liczbę rzeczywistą
long()	Konwersja na liczbę typ long
complex()	Konwersja na liczbę zespoloną
str()	Konwersja na łańcuch znakowy

Aktualny typ zmiennej można sprawdzić przy użyciu funkcji *type()*.

Listing 1.2. Silne typowanie w Pythonie

```
>> foo = 1
>> bar = '2'
>> foobar = foo + bar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>foobar = foo + int(bar)
>>foobar
2
```

Zarówno na liczbach jak i stringach można wykonywać proste operacje. Podstawowe operatory przedstawiono w tabeli 1.2

Listing 1.3. Proste operacje w języku Python

```
>>foo = 1
>>bar = 2
>>foobar = foo + bar
>>foobar
3
>>qux = 'Lorem'
>>quux = 'ipsum'
>>quuz = qux + ' ' + quux
>>quuz
Lorem ipsum
>>spam = 1j
>>eggs = spam ** 2
>>eggs
(-1+0j)
>>corge = 'AB'
>>grault = corge * 4
>>corge
'ABABABAB'
```

Tabela 1.2. Podstawowe operacje

Operator	**Opis**
+	Dodawanie
-	Odejmowanie
*	Mnożenie
/	Dzielenie
%	Reszta z dzielenia (modulo)
**	Potęgowanie

Ciągi znakowe w Pythonie można formatować z wykorzystaniem operatora %. Stosując zapis:

format % wartości

elementy formatujące pola format zostaną zastąpione elementami pola wartość. Jeżeli w polu format użyto jednego argumentu (zmiennej) to pole wartość jest pojedynczym elementem – tą zmienną. W przeciwnym razie jest krotką składającą się z odpowiednich elementów. Przykłady formatowania ciągów znakowych przedstawiono na listingu poniżej.

Listing 1.4. Formatowanie ciągów znakowych

```
>>foo = 1
>>bar = 'foo'
>>foobar = 'Wartosc zmiennej %s wynosi %d' % (bar,foo)
>>foobar
'Wartosc zmiennej foo wynosi 1'
>>qux=124
>>quux = 'Liczba dziesietna %d to szestnastkowo %X' % (qux,qux)
>>quux
'Liczba dziesietna 124 to szestnastkowo 7C'
>>spam = 'Numery:'
>>ham = 1.34
>>eggs = 2
>>spam = '%s %5.4f %02d' % (spam,ham,eggs)
>>spam
'Numery: 1.3400 02'
```

1.2. *Listy, słowniki, tuple*

Listy w Pythonie to nic innego jak zbiory rożnych elementów. Elementami listy mogą być wszystkie typy danych. Ponadto listy mogą być dowolnie zagnieżdżane. Na listingu 1.5 przedstawiono sposób definiowania list oraz dostępu do ich poszczególnych elementów. Warto zwrócić uwagę że użycie ujemnego indeksowania pozwala na odniesienie do elementu względem końca listy. Jak widać listy mogą zawierać również elementy rożnych typów.

Listing 1.5. Tworzenie list i dostęp do elementów

```
>>baz = ['Poniedziałek', 'Wtorek', 'Środa', 'Czwartek', 'Piątek']
>>baz
['Poniedziałek', 'Wtorek', 'Środa', 'Czwartek', 'Piątek']
>>baz[0]
'Poniedziałek'
>>baz[-2]
'Czwartek'
>>qux = ['Jeden', [1, 2], 3+0j]
>>qux[1][1] #Odczyt zagnieżdżonego elementu listy
2
```

Do utworzenia listy w postaci ciągu arytmetycznego możemy użyć polecenia *range()*. Zakresu możemy użyć dwojako. Używając formy:

range(n)

Wtedy zostanie utworzona n-elementowa lista kolejnych liczb całkowitych rozpoczynając od zera. Polecenia range możemy również użyć jako:

range(x1,x2,dx)

gdzie odpowiedni:

x1 – początkowy element listy,

x2 – końcowy element listy,

dx – krok (różnica między kolejnymi elementami)

Na listingu 1.6 przedstawiono przykłady tworzenia list będących ciągami arytmetycznymi oraz zawierających wiele tych samych elementów.

Listing 1.6. Tworzenie list cd.

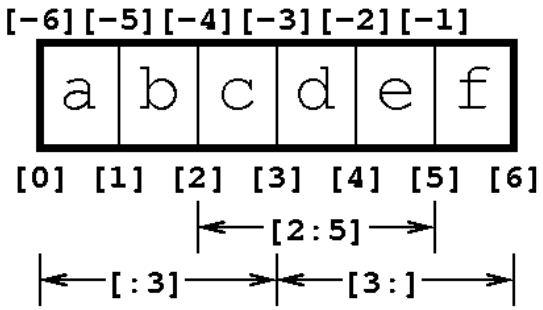
```
>>range(5)
[0, 1, 2, 3, 4]
>>range(2,7)
[2, 3, 4, 5, 6]
>>range(2,7,2)
[2, 4, 6]
>>[1]*3
[1, 1, 1]
>>foo = [1, 2, 3]
>>foobar = foo * 2
>>foobar
[1, 2, 3, 1, 2, 3]
```

W Pythonie w łatwy sposób możemy używać „wycinków” list. Możemy dokonywać operacji bezpośrednio tylko na określonym zakresie listy ale również przepisywać fragmenty list i w ten sposób tworzyć nowe. Sposób dostępu do poszczególnych elementów list przedstawia rysunek 1.1.

Listing 1.7. „Wycinki list”

```
>>foo=[10, 11, 12, 13, 14]
>>foo[0:2]
[10, 11]
>>foo[:2] #skrócony zapis j/w
```

```
[10, 11]
>>foo[-1]
14
>>bar = foo[1:3]
>>bar
[11, 12]
```



Rysunek 1.1. Dostęp do poszczególnych elementów list

Przydatnym narzędziem w języku Python jest tzw. wytwornik list. Wytworniki list tworzy się według następującego schematu:

[*wyrażenie* for *zmienna* in *sekwencja*)

Poniżej przedstawiono przykłady użycia wytworników list.

Listing 1.8. Przykłady użycia wytworników list

```
>>[(2*x+3) for x in range(5)]
[3, 5, 7, 9, 11]
>>[(x,x**3) for x in range(2,5)]
[(2, 8), (3, 27), (4, 64)]
>>foo=range(7)
>> [x for x in foo if x > 4] #Wytwornik listy z warunkiem
[5, 6]
```

Listy będące w Pythonie obiektami posiadają szereg metod umożliwiających operacje na nich. Metody umożliwiające operacje na listach przedstawiono w tabeli 1.3. Przykłady ich wykorzystania zostały zawarte na listingu 1.9.

Tabela 1.3. Metody umożliwiające operacje na listach

Metoda	**Opis**
Append(x)	Dodanie elementu x na końcu listy
Count(x)	Zliczenie wystąpienia wartości x w liście
Index(x)	Indeks pierwszego wystąpienia elementu x na liście

Remove(x)	Usunięcie pierwszego elementu o wartości x z listy
Sort()	Sortowanie listy
Reverse()	Odwrócenie kolejności elementów listy

Listing 1.9. Operacje na listach

```
>>spam = [10, 21, 23, 11, 24]
>>spam.append(11)
>>print spam
[10, 21, 23, 11, 24, 11]
>>spam.count(11)
2
>>spam.index(11)
3
>>spam.remove(11)
>>print spam
[10, 21, 23, 24, 11]
>>spam.sort()
>>print spam
[10, 11, 21, 23, 24]
>>spam.reverse()
>> print spam
[24, 23, 21, 11, 10]
```

Kolejnym typem danych w języku Python są słowniki. Dostępu do elementów słownika nie dokonujemy jak w przypadku list z użyciem indeksu lecz przez podanie klucza. Słownik zatem to typ sekwencji który składa się z kluczy i przyporządkowanych do nich wartości. Tworzenie słowników i operacje na nich przedstawiono na listingu poniżej.

Listing 1.10. Tworzenie słowników i operacje na nich

```
>>foo = {'imie':'Nikola', 'nazwisko':'Tesla'}
>>foo['imie']
'Nikola'
>>foo['rok_ur'] = 1856 #Dodanie elementu do słownika
>> print foo
{'imie':'Nikola', 'nazwisko':'Tesla', 'rok_ur':1856}
>>bar ={} #Utworzenie pustego słownika
```

Przedstawione wyżej sekwencje danych (listy, słowniki) umożliwiają modyfikacje ich zawartości. Niemodyfikowalną sekwencją obiektów w Pythonie są krotki (tuple). Przykład użycia tupli przedstawiono na listingu 1.11

Listing 1.11. Użycie tupli w Pythonie

```
>>foobar = (1, 'dwa')
>>print foobar
```

```
(1, 'dwa')
>>foobar[0]
1
```

1.3. Instrukcje warunkowe, pętle

W Pythonie warunkowego wykonywania operacji dokonuję się z wykorzystaniem konstrukcji if-(elif)-(else):

if <wyrażenie warunkowe>:
    instrukcje
elif <wyrażenie warunkowe>:
    instrukcje
else:
    instrukcje

Do wykonania porównania w języku Python wykorzystuje się dwa znaki równości (==). Wyrażenie „nie równe" w Pythonie oznacza się jako **!=**. Do łaczenia wyrażeń warunkowych używa się operatorów **or** oraz **and** które odpowiadają logicznym operacją boolowskim. Na listingu 1.12 przedstawiono przykładowe wyrażenie warunkowe.

Listing 1.12. Przykładowe wyrażenie warunkowe w Pythonie

```
spam = 1.4
if spam > 2:
      ham = 1
elif spam > 1 and spam < 2:
      ham = 2
else:
      ham = 3
```

W języku Python istnieje również trójargumentowe wyrażenie warunkowe:

<wyrażenie1> if <wyrażenie warunkowe> else <wyrażenie2>

Jest ono odpowiednikiem konstrukcji warunkowej z operatorem **?** z języka C.

Podobnie jak w innych językach programowania wysokiego poziomu, również w Pythonie możliwe jest iteracyjne (cykliczne) wykonywanie operacji z wykorzystaniem pętli. Pętla for wykorzystywana do iteracyjnego wykonywania instrukcji ma następującą postać:

for <pozycja> in <sekwencja>:
    instrukcje

Przy stosowaniu pętli for pomocna jest funkcja *range()* tworząca sekwencje ciągu liczb całkowitych. Przykładowe użycie pętli for przedstawiono na listingu poniżej.

Listing 1.13. Przykładowe użycie pętli for

```
for foo in 'PYTHON':
      print '%s.' % (foo)
foobar = []
for bar in range(5):
      foobar.append(bar)
```

Pętlą while wykonująca się do czasu gdy spełnione jest wyrażenie warunkowe ma postać:

while <wyrażenie warunkowe>:
    instrukcje

Przykładowe użycie pętli while przedstawiono poniżej.

Listing 1.13. Przykładowe użycie pętli while

```
while(True): #Pętla nieskończona
      pass

foo = 1
while(foo < 5):
      print foo
      foo += 1
```

Warto zwrócić uwagę że powyżej została użyta instrukcja pass. Nie robi ona nic konkretnego – używa się jej tam gdzie składnia wymaga obecności instrukcji.

1.4. Funkcje, klasy i obiekty

Język Python umożliwia tworzenie funkcji i korzystanie tym samym z funkcyjnego paradygmatu programowania. Funkcje tworzy się według następującego wzoru:

def **nazwa_funkcji**(<argumenty>):
    instrukcje
    return <wyrażenie zwracane przez funkcje>

Ponadto możliwe jest również definiowanie funkcji anonimowych z wykorzystaniem operatora lambda:

<funkcja> = lambda <argumenty> : <wyrażenie>

Przykładowe funkcje przedstawiono na listingu 1.14

Listing 1.14. Tworzenie funkcji w języku Python

```
def Pierwiastek(foo,bar):
      return foo ** (1 / bar)

def Dodaj(spam,ham):
      eggs = spam + ham
      return eggs

def KodASCII(foobar):
      return ord(foobar)

Dodaj2 = lambda foo,bar : foo+bar
```

Python jest językiem zorientowanym obiektowo – umożliwia tworzenie klas, dziedziczenie ze stworzonych i wbudowanych klas oraz tworzenie obiektów. Klasy w języku Python tworzymy według następującego schematu:

class n**azwa_klasy**(<klasy z których klasa dziedziczy>):
atrybuty
metody

W Pythonie istnieją takie metody które maja specjalne znaczenie. Jedną z nich jest metodą o nazwie ***___init___()*** która jest wywoływana automatycznie po utworzeniu obiektu. Metoda ta często błędnie nazywana jest konstruktorem – w momencie jej wywoływania obiekt już istnieje. Przykład tworzenia klasy oraz jej instancji (obiektu) przedstawiono poniżej.

Listing 1.15. Tworzenie klasy i obiektu

```
class Szescian():
      def __init__(self, foo):
            self.bar = foo
      def PolePow(self):
            return 6 * (self.bar ** 2)
      def Objetosc(self):
```

```
            return self.bar ** 3

...

>>szescian1 = Szescian(2)
>>print szescian1.PolePow()
24
```

W Pythonie pierwszy argument metody należącej do danej klasy (referencja do instancji tej klasy) nazwany jest **self**. Argument ten pełni tą samą rolę co np. słowo **this** w języku C++ jednak nie jest to zastrzeżone słowo a jedynie umowną koncepcją nazewnictwa.

2. Moduł NumPy

NumPy to podstawowy moduł umożliwiający wykonywanie zaawansowanych obliczeń matematycznych w języku Python. Umożliwia dokonywanie operacji na wektorach i macierzach.

2.1. Importowanie modułów w Pythonie

Aby móc korzystać z bibliotek należy je zaimportować. W Pythonie istnieją dwa sposoby importowania modułów. Pierwszy sposób to zaimportowanie modułu z wykorzystaniem instrukcji:

import <moduł> <as> <skrócona nazwa modułu>

Drugi sposób korzysta z konstrukcji:

from <moduł> import <atrybuty,metody>

Może się wydawać, że użycie polecenia `import numpy` oraz `from numpy import *` (gwiazdka oznacza że importujemy wszystko) da taki sam efekt. Istnieje jednak zasadnicza różnica która widoczna jest podczas dostępu do atrybutów i metod danego modułu. W przypadku użycia konstrukcji `from <moduł> import <atrybuty,metody>` importowanie zostaje wykonane do lokalnej przestrzeni nazw. W tym przypadku dostęp do atrybutów i metod jest możliwy bezpośrednio bez podawania modułu z którego pochodzą – listing 2.1.

Listing 2.1. Importowanie bibliotek w języku Python

```
>>import numpy as np
>>pi
Traceback (most recent call last):
```

```
  File "<ipython-input-2-68f7b1e53523>", line 1, in <module>
    pi

NameError: name 'pi' is not defined
>>np.pi
3.141592653589793
>>from numpy import *
>>pi
3.141592653589793
```

2.2. Tworzenie macierzy

W module Numpy wyróżnić można dwa podstawowe typy macierzowe:

- matrix
- array

Najistotniejsza różnica między typami macierzowymi w module NumPy to sposób wykonywania operacji mnożenia.

1. W przypadku tablic (array) użycie operatora * powoduje mnożenie elementowe macierzy – odpowiednik operacji .* w MATLAB-ie. Aby wykonać mnożenie macierzy (a nie ich elementów) należy użyć metody *dot()*.
2. Używając obiektów klasy matrix operator * powoduje mnożenie macierzy. Mnożenie elementów macierzy możliwe jest z wykorzystaniem metody *multiply()*

Przykłady tworzenia tablic i macierzy oraz wspomniane wyżej różnice w wykonywaniu operacji mnożenia przedstawiono na listingu 2.2. Należy zwrócić uwagę, że indeksowanie elementów zaczyna się od zera (nie jak w MATLAB-ie od 1).

Listing 2.1. Rożnicę między typami macierzowymi w NumPy

```
>>A = np.array([[1, 2], [3, 4]])
>>B = np.array([[1, 1], [2, 2]])
>>A * B
array([[1, 2],
       [6, 8]])
>>np.dot(A,B)
array([[ 5,  5],
       [11, 11]])
>>A= np.matrix([[1, 2], [3, 4]])
>>B = np.matrix([[ 1, 1], [2, 2]])
>>A * B
matrix([[ 5,  5],
        [11, 11]])
>>np.multiply(A,B)
```

```
matrix([[1, 2],
        [6, 8]])
>>A[0,0]
1
```

Aby utworzyć macierz w postaci wektora stanowiącego ciąg arytmetyczny można wykorzystać następujące metody:

arange(n)

arange(x1,x2,k)

linspace(x1,x2,n)

gdzie:

x1- początkowy element tablicy

x2 – końcowy element tablicy

k – krok – różnica między kolejnymi elementami ciągu

n – liczba elementów wektorach

Aby uzyskać wektor elemntów o rozkładzie w skali logarytmicznej należy użyć metody *logspace()*. Jej użycie jest analogiczne do metody *linspace()*. Ponadto moduł NumPy zawiera szereg metod pozwalających szybkie tworzenie „typowych macierzy" takich jak:

- ones – macierz zer
- zeros – macierz jedynek
- eye – macierz jednostkowa

Ich użycie przedstawiono na listingu 2.3.

Listing 2.3. Tworzenie macierzy - NumPy

```
>>Z = zeros((2,4),uint8)
>>print Z
array([[0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=uint8)
>>I = eye(3)
>>print I
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>A = np.arange(9)
```

```
>>A
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>A.shape = (3,3) #Formowanie kształtu macierzy
>>A
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>A.T #Transponowanie tablicy
array([[0, 3, 6],
       [1, 4, 7],
       [2, 5, 8]])
>>A.min() #Wartosc minimalna w tablicy
0
>>A.size #Ilość elementów macierzy
9
```

Powyżej przedstawiono przykładowe atrybuty i metody obiektu typu array. Szczegółową dokumentację klasy ndarray znaleźć można w SciPy.org.

2.3. ***Podstawowe operacje matematyczne, operacje porównania***

Operacje matematyczne na tablicach możliwe są z wykorzystaniem operatorów lub specjalnych funkcji. Przykładowe użycie przedstawiono na listingu poniżej. Funkcje matematyczne przedstawiono w tabeli 2.1.

Listing 2.4. Operacje matematyczne na macierzach - NumPy

```
>>A = np.array((1, 2, 3))
>>B = np.array((2, 3, 5))
>>A+B
array([3, 5, 8])
>>np.add(A,B)
array([3, 5, 8])
>>np.power(A,2)
array([1, 4, 9])
```

Tabela 2.1. Operatory i funkcje matematyczne

Operator	**Funkcja**
A+B	add(A,B)
A-B	subtract(A,B)
A%B	remainder(A,B)
A*B	multiply(A,B)

A/B	divide(A,B)
A ** B	power(A,B)
	dot(A,B)

Biblioteka NumPy umożliwia również porównywanie zawartości tablic oraz dokonywania operacji logicznych. Przykładowe operacje przedstawiono na listingu 2.5. Podobnie jak w przypadku operacji matematycznych mozliwe jest wykorzystanie operatorów jaki i specjalnych funkcji.

Listing 2.5. Operacje porównania i logiczne - przykłady

```
>>A = np.array((1, 2, 3, 4))
>>B = np.array((1, 3, 5, 4))
>>A == B
array([ True, False, False,  True], dtype=bool)
>>np.equal(A,B)
array([ True, False, False,  True], dtype=bool)
>>np.any(A > 6) #Prawda, jeżeli któryś z elementów spełnia warunek
False
>>np.any(A > 3)
True
>>np.all( A > 0 ) #Prawda, jeżeli wszystkie z elementów spełniają warunek
True
>> np.all( A > 0 ) and np.all( A > 3 )
False
```

Tabela 2.2. Operatory porównania i logiczne

Operator	**Funkcja**
A == B	equal(A,B)
A >= B	greater_equal(A,B)
A <= B	less_equal(A,B)
A > B	greater(A,B)
A < B	less(A,B)
A != B	not_equal(A,B)
<wyrażenie1> and <wyrażenie2>	logical_and(<wyrażenie1>,<wyrażenie2>)
<wyrażenie1> or <wyrażenie2>	logical_or(<wyrażenie1>,<wyrażenie2>)
not <wyrażenie1>	logical_not(<wyrażenie>)
	logical_xor(<wyrażenie>)

2.4. Funkcje trygonometryczne, statystyczne oraz inne funkcje matematyczne

Biblioteka NumPy zawiera szereg funkcji matematycznych. Dostępne są m.in. funkcje trygonometryczne, funkcje umożliwiające proste działania statystyczne oraz szereg innych funkcji. Ważniejsze funkcje przedstawiono w tabeli 2.3.

Tabela 2.2. Funkcje matematyczne - NumPy

Funkcja	**Opis**
sin(x)	Sinus kąta x
sinh(x)	Sinus hiperboliczny z x
cos(x)	Cosinus kąta x
cosh(x)	Cosinus hiperboliczny x
arcsin(x)	Arcus sinus x
arcsinh(x)	Arcus sinus hiperboliczny x
arccos(x)	Arcus cosinus x
arccosh(x)	Arcus cosinus hiperboliczny x
arctan(x)	Arcus tangens x
arctanh(x)	Arcus tangens hiperboliczny x
arctan2(x,y)	Zwraca kąt którego tangens jest ilorazem x oraz y
exp(x)	Funkcja ekspotencjonalna - e^x
log(x)	Logarytm naturalny z x
log10(x)	Logarytm dziesiętny z x
sqrt(x)	Pierwiastek kwadratowy z x
absolute(x)	Wartość bezwzględna z x
conjugate(x)	Sprzężenie liczby zespolonej x
negative(x)	Liczba odwrotna do x
ceil(x)	Zaokrąglenie liczby x w górę
floor(x)	Zaokrąglenie liczby x w dół
fabs(x)	Wartość bezwzględna liczby zmiennoprzecinkowej x
hypot(x,y)	Funkcja odległości euklidesowej x i y ($\sqrt{x^2+y^2}$)

fmod(x,y)	Reszta z dzielenia liczby rzeczywistych x i y
maximum(x,y)	Wartość maksymalna z liczb x i y
minimum(x,y)	Wartość minimalna z liczb x i y

Przykładowe wykorzystanie wyżej przedstawionych funkcji pokazano na listingu poniżej.

Listing 2.6. Funkcje matematyczne – biblioteka NumPy

```
>>foo = sin(2) + exp(3)
>>foo
20.994834350013349
>>bar = ceil(foo)
>>bar
21.0
>>negative(bar)
-21.0
>>fmod(2.3, 1.4)
0.89999999999999991
>>foobar = 3+2j
>>conjugate(foobar)
(3-2j)
```

3. Biblioteka Matplotlib

Matplotlib to biblioteka do tworzenia wykresów stworzona dla języka Python i modułu NumPy. API biblioteki zostało zaprojektowane w ten sposób aby jak najbardziej przypominało to z MATLAB-a. Szczegółowa dokumentacja biblioteki dostępna jest na stronie projektu. W tabeli 3.1 przedstawiono podstawowe funkcje umożliwiające tworzenie wykresów 2D.

Tabela 2.2. Podstawowe funkcje biblioteki Matplotlib do rysowania wykresów

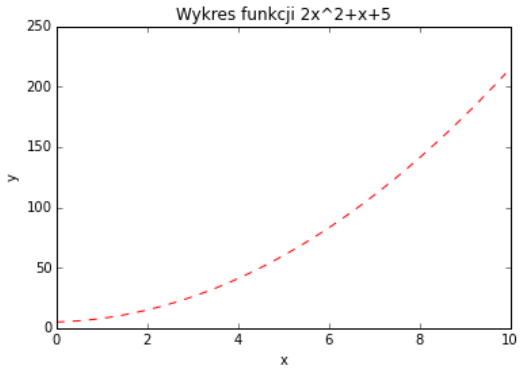
Funkcja	**Opis**
figure(x)	Otwarcie okna rysunkowego x
plot(x,y,styl_linii)	Stworzenie wykresu 2D funkcji y=f(x)
show()	Wyświetlenie obrazu
title('tekst')	Dodanie tytułu do wykresu
xlabel('tekst')	Dodanie opisu osi x
ylabel('tekst')	Dodanie opisu osi y

axis([xmin, xmax, ymin, ymax])	Ograniczenie osi wykresu w podanym zakresie
grid(True/False)	Włączenie/wyłączenie siatki
savefig('nazwa_pliku')	Zapis rysunku do pliku
legend('tekst')	Dodanie legendy do wykresu
hold(True/False)	Zablokowanie okna rysunkowego
subplot(l.wierszy,l.kolumn,nr_okna)	Utworzenie podwykresu
[X,Y] =meshgrid(x,y)	Przekształca parę wektorów x,y w parę macierzy X,Y – do wyznaczania funkcji 2 zmiennych
contour(x,y,z)	Stworzenie wykresy konturowego funkcji y=f(x,y)
contourf(x,y,z)	Wypełniony wykres konturowy funkcji y=f(x,y)
surf(x,y,z)	Stworzenie wykresu 3D funkcji y=f(x,y)
specgram(x)	Stworzenie spektogram x
imshow(A)	Wyświetlenie obrazu z macierzy A

Przykład prostego wykresu stworzonego z wykorzystaniem biblioteki przedstawiono poniżej.

Listing 3.1. Wykres funkcji $2x^2+x+5$ - Matplotlib

```
from pylab import *
x=arange(0, 100, 0.1) #stworzenie wektora wartości x
y=2 * (x ** 2) + x + 5 #wyznaczenie wartości y
plot(x,y,'r-') #Narysowanie wykresu y=f(x)
title('Wykres funkcji 2x^2+x+5')
xlabel('x')
ylabel('y')
savefig('wykres.png') #Zapis wykresu do pliku
```



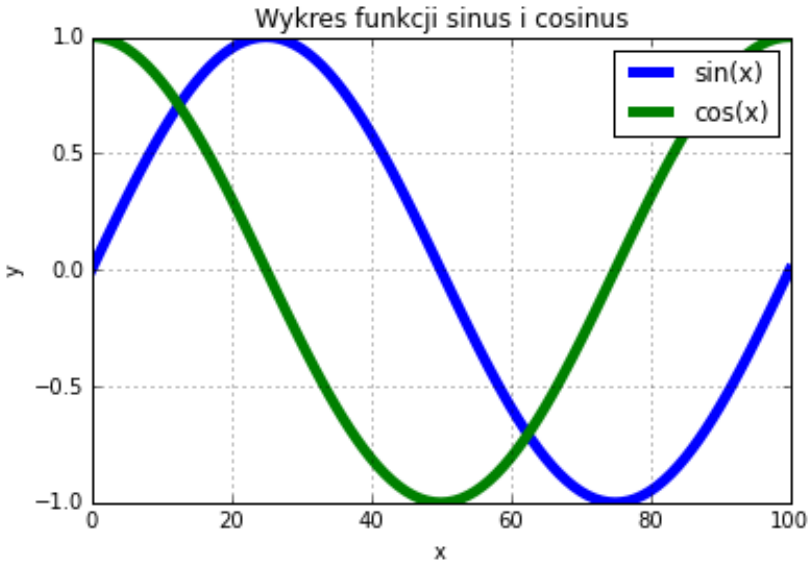
Rysunek 3.1. Efekt działania programu z listingu 3.1.

Przykłady wykorzystania pakietu NumPy i biblioteki Matplotlib

Przykład 1

Listing 3.2. Program przykładowy nr 1

```
from scipy import *
from pylab import *
x = r_[0:101]
y01 = sin( 2 * pi * x  /100)
y02 = cos( 2 * pi * x / 100)
plot(x, y01, linewidth = 5.0)
hold(True)
plot(x, y02, linewidth = 5.0)
xlabel('x'); ylabel('y')
title('Wykres funkcji sinus i cosinus');
legend(('sin(x)','cos(x)'));
grid(True)
```



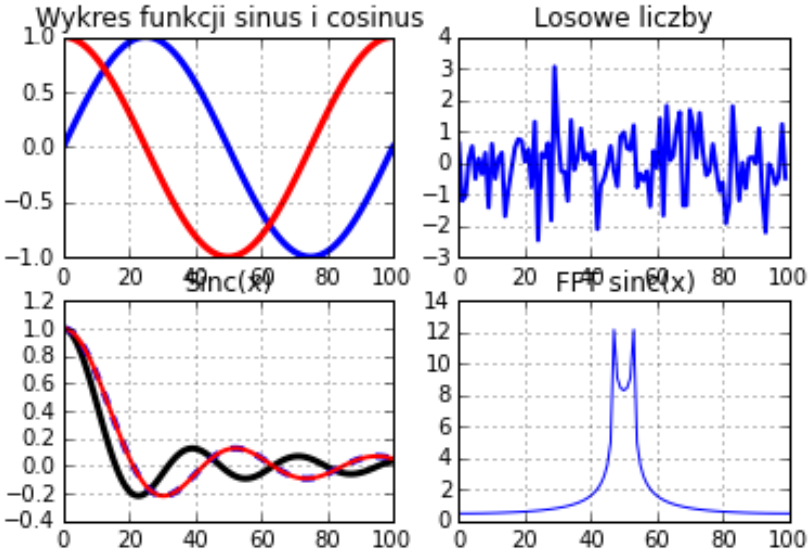
Rysunek 3.2. Wynik działania program przykładowego 1

Przykład 2

Listing 3.3. Program przykładowy nr 2

```
from pylab import *
from scipy import *
from scipy.fftpack import fftshift
x = r_[0:101]
y01 = sin(2 * pi * x / 100)
y02 = cos(2 * pi * x / 100)
y03 = randn(100);
```

```
y04 = sinc(2 * pi * x / 100);
Y04 = abs(fftshift(fft(y04)))
y05 = sinc(1.5 * pi * x /100);
y06 = sinc(1.5 * pi * x /100);
Y06 = abs(fftshift(fft(y06)))
subplot(2,2,1); plot(x, y01, linewidth =3);
hold(True);
plot(x,y02,'r', linewidth =3);
grid(True); title('Wykres funkcji sinus i cosinus');
subplot(2,2,2); plot(y03,linewidth =2);
grid(True);
title('Losowe liczby');
subplot(2,2,3); plot(x,y04,'k', linewidth=3);
grid(True); title('Sinc(x)');
hold(True);
subplot(2,2,3); plot(x,y05,'--', linewidth=3);
subplot(2,2,3); plot(x,y06,'r', linewidth =2);
subplot(2,2,4); plot(Y04); grid(True);
title('FFT sinc(x)');
show()
```



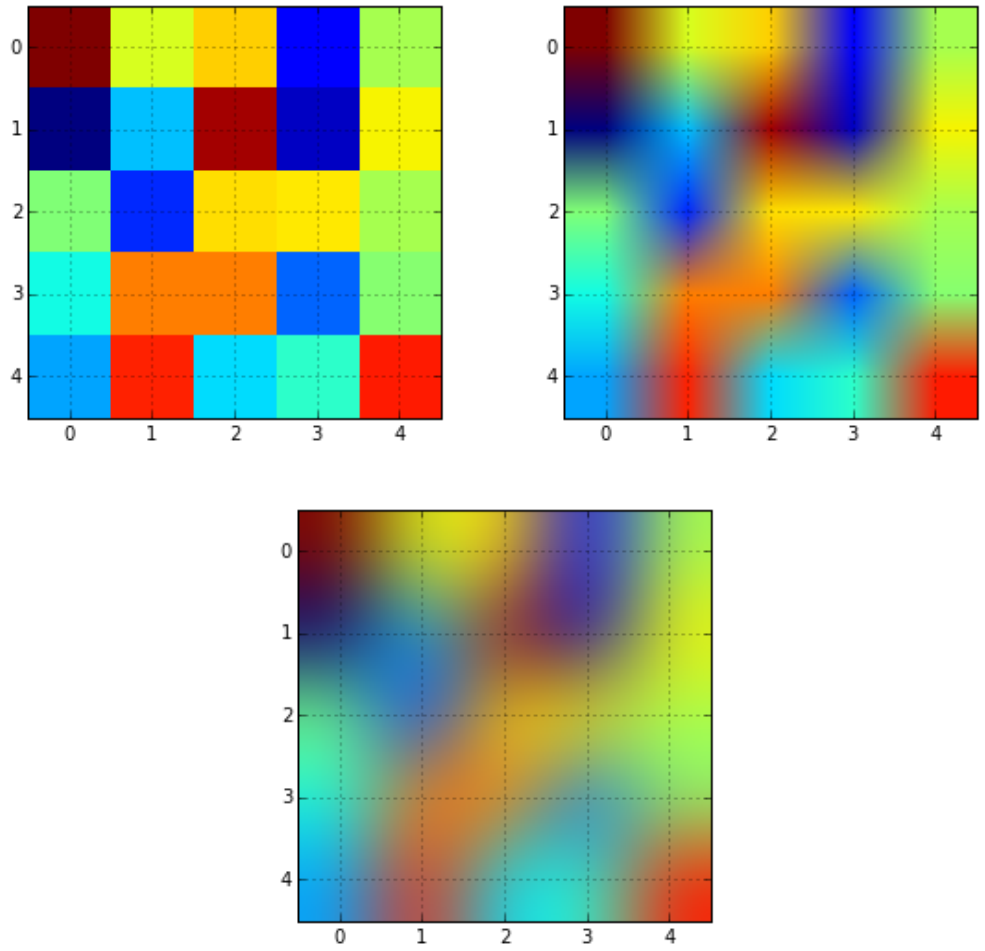
Rysunek 3.3. Wynik działania program przykładowego 2

Przykład 3

Listing 3.3. Program przykładowy nr 3

```
from pylab import *
A = rand(5,5)
figure(1)
imshow(A,interpolation='nearest')
grid(True)
figure(2)
imshow(A, interpolation='bilinear')
```

```
grid(True)
figure(3)
imshow(A,interpolation='bicubic')
grid(True)
show()
```



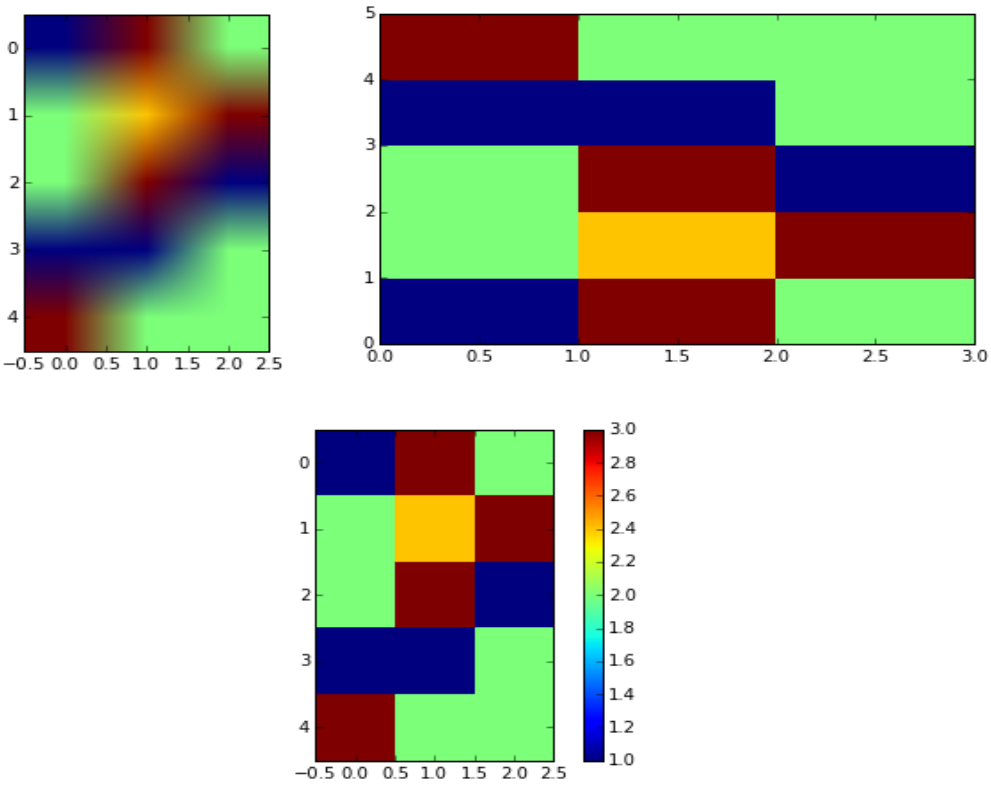
Rysunek 3.4. Wynik działania program przykładowego 3

Przykład 4

Listing 3.3. Program przykładowy nr 4

```
import pylab as p
X = p.array([[1, 3, 2],
             [2, 2.4, 3],
             [2, 3, 1],
             [1, 1, 2],
             [3, 2, 2]])
p.figure()
p.imshow(X)
p.figure()
```

```
p.pcolor(X)
p.figure()
p.imshow(X, interpolation='nearest')
p.colorbar()
p.show()
```



Rysunek 3.4. Wynik działania program przykładowego 4

Zadania do wykonania

1. Zapoznać się z podstawami języka Python, uruchomić i zmodyfikować przykłady z instrukcji
2. Zapoznać się z modułem NumPy. Uruchomić i zmodyfikować przykłądy z instrukcji.
3. Zapoznać się z biblioteką Matplotlib. Uruchomić przykłady z instrukcji. Utworzyć własne wykresy.